TURING 图灵程序设计丛书

日本亚马逊2012年度销量No.1



JavaScript编程全解

对应 ECMAScript 最新版本

完美涵盖AJAX / jQuery / HTML5 / WebSocket / Web API / Node.js

【日】井上诚一郎 土江拓郎 滨边将太 著
陈筱烟 译

人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明



井上诚一郎

曾在美国参与过Lotus Notes的开发，后在日本创立了Ariel Network股份公司，任CTO。目前从事面向企业的PSP软件及企业产品的开发。著有《PSP教科书》、《Java编程详解》、《实践JS：服务器端JavaScript入门》等书。

土江拓郎

2008年加入Ariel Network股份公司。从事Java及JavaScript相关的企业产品开发工作。

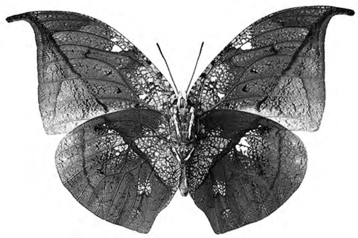
滨边将太

2009年加入雅虎公司，从事针对电视的软键盘开发，以及智能手机应用GyaO!的开发。

陈筱烟

毕业于复旦大学计算机科学与技术系，主要研究方向为跨设备人机交互理论。长期从事对日软件外包工作。从大学时期开始接触并使用Java、JavaScript进行程序开发，现在对Web应用及智能手机应用的开发很感兴趣。

图灵程序设计丛书



JavaScript编程全解

【日】井上诚一郎 土江拓郎 滨边将太 著
陈筱烟 译

人民邮电出版社
北 京

图书在版编目(CIP)数据

JavaScript编程全解 /(日)井上诚一郎,(日)土江拓郎,(日)滨边将太著;陈筱烟译. -- 北京:人民邮电出版社,2013.12

(图灵程序设计丛书)

ISBN 978-7-115-33341-4

Ⅰ. ①J… Ⅱ. ①井… ②土… ③滨… ④陈… Ⅲ. ①JAVA语言—程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2013)第240740号

内 容 提 要

本书涵盖了JavaScript开发中各个方面的主题,对从客户端及服务器端JavaScript等基础内容,到HTML5、Web API、Node.js与WebSocket等热门技术,都作了深入浅出的介绍与说明。读者能够通过本书了解当今JavaScript开发的最新现状。本书的一大特色是对JavaScript语言的语法规则进行了细致的说明,并通过大量纯正的JavaScript风格代码,帮助读者准确地掌握JavaScript的语言特性及细节用法。

本书适合JavaScript开发初学者系统入门、有经验的JavaScript开发者深入理解语言本质,也适合开发团队负责人、项目负责人作为综合性的JavaScript参考书阅读。

◆ 著　[日]井上诚一郎　土江拓郎　滨边将太
译　陈筱烟
责任编辑　乐　馨
执行编辑　徐　骞
责任印制　焦志炜

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　印刷

◆ 开本:800×1000　1/16
印张:26.25
字数:794千字　2013年12月第1版
印数:1-3 000册　2013年12月北京第1次印刷

著作权合同登记号　图字:01-2013-3125号

定价:79.00元

读者服务热线:(010)51095186转600　印装质量热线:(010)81055316

反盗版热线:(010)81055315

广告经营许可证:京崇工商广字第0021号

版 权 声 明

注意

购买、使用本书前的必读事项

- 本书所记述的内容，仅仅是作者向读者提供的信息，因此，读者对本书内容的使用，基于其自身的判断，一切责任自负。对于使用本书内容所造成的结果，技术评论社、原书作者、人民邮电出版社以及译者，概不负责。

- 本书内容依据 2011 年 8 月 30 日的情况所写，因此在阅读本书时，个别内容可能已经发生了更改。对于部分变更，已经附上了译者注，帮助读者理解。关于书中出现的软件信息，如无特别说明，均以 2011 年 8 月 30 日的最新版为准。在软件版本升级之后，其功能与界面可能会与本书中的说明有所差异。在购买本书前，请务必确认软件版本号。

- 本书内容及其收录的范例代码，已确认能够在以下环境中运行。

操作系统	Windows 7 Professional 64位版
浏览器	Internet Explorer／ Firefox／ Google Chrome

在除此之外的环境中使用时，操作方法、软件界面以及程序的执行方式可能会与本书的记述有所不同，敬请谅解。

请在理解以上这些注意事项的基础上使用本书。

- 可以在下面的站点中找到本书的支持信息(日文)。

  http://gihyo.jp/book/2011/978-4-7741-4813-7/support

前　　言

首先感谢您购买本书。

这是一本关于 JavaScript 程序设计语言的书。本书的前半部分将对 JavaScript 的语言基础进行解说，而后半部分主要介绍包括客户端 JavaScript、HTML5、Web API 以及服务器 JavaScript 等与 JavaScript 相关的应用领域。

本书面向有一定编程基础的开发者，因此书中的 JavaScript 代码都只取了片段。希望通过复制粘贴来使用这些代码的读者，在阅读本书时或许会感到有些困难。此外，本书中没有涉及网页设计和用户体验的内容，如果只是希望学习和网页显示效果有关的 JavaScript 知识，阅读本书并不合适。

本书的目标读者是希望深入学习 JavaScript 并开发完整的 Web 应用程序的人。那些平时主要使用 Java 或是 PHP 等其他语言的开发者，有时也会遇到一些不得不使用 JavaScript 语言的情况。对于这些开发者来说，如果你抱有"既然要使用 JavaScript，就应该学习正确的语言规范以写出良好代码"的想法，也推荐你阅读本书。

如果希望理解 JavaScript 的语言基础，请阅读本书的第 2 部分。为了让没有接触过程序设计语言的人也能理解，本书着实下了一番功夫。虽然其中包含了不少和 Java 对比的内容，不过只要按顺序认真阅读，初学者也不会感到吃力。

从第 3 部分开始本书将介绍 JavaScript 的应用，其中还包括 HTML5 和 Node.js 等热门的新技术。其实，如果真要理性地评价 JavaScript 这门程序设计语言，我们会发现它并不具有令人兴奋的特性。虽然 JavaScript 在其看似平凡的外表之下有着复杂的内部构造，但它确实不是一种采用了计算机科学领域最新技术的新型语言。事实上，学习 JavaScript 的意义与其说是为了学习这门语言本身，倒不如说是为了学习和使用其相关领域的知识。近年来，JavaScript 相关领域的发展着实令人着迷。毫不夸张地说，现在互联网的新热点大多都和 JavaScript 有关。如果本书的后半部分内容能够将这种兴奋感传达给各位读者的话，我将感到不胜荣幸。

井上诚一郎

2011 年 8 月 31 日

■目标读者

- 读过一些 JavaScript 的入门书籍，希望进一步了解 JavaScript 本质的人。
- 平时虽然常常使用 JavaScript，但还没有完全理解 JavaScript，并因此感到有些信心不足的人。
- 虽然主要使用其他的程序设计语言，但有时也会使用 JavaScript 的人。
- 认为 JavaScript 会是今后的主流语言的人。

目录 CONTENTS



第 1 部分

JavaScript 概要

本书首先介绍 JavaScript 的现状、语言特性，以及与其相关的一些领域。

第1章 JavaScript概要

本章将介绍JavaScript和ECMAScript的关系与历史，以及JavaScript与作为其实现方式和运行环境的浏览器的关系，此外还将总括JavaScript的可移植性。

1.1 JavaScript概要

我们首先介绍JavaScript相关的运行环境，其语言特征会在第2部分详述。正在读本书的读者，应该都知道JavaScript是在浏览器中运行的语言吧。甚至可以说，除开发者以外，被大众所熟知的程序设计语言也许只有JavaScript。而且在软件史上，以能够在各种环境下运行而著称的语言中，大概没有比JavaScript更有名的了。

但是，正是由于太过常见，才让很多人对JavaScript有了一些误解与偏见。

例如，因为和浏览器的关联性过强，很多人都以为JavaScript只能在浏览器中运行。对JavaScript的看法也是莫衷一是。有人认为它降低了Web的使用体验，也有人称赞它是一门使Web的易用性得以进化的出色的技术。有人觉得JavaScript是任何人都可以学会的简单语言，也有人认为它过于抽象，很难掌握。

对JavaScript的看法各有不同，很难说哪一种正确。不过，只要软件以Web为中心，今后JavaScript的重要性就一定会进一步提升。JavaScript领域的名人道格拉斯•克罗克福德曾把JavaScript称为Web上的虚拟机。其核心含义是，在JavaScript广为普及的现在，Web已经成为了JavaScript事实上的运行环境。夸张地讲，JavaScript正日益成为支配世界的程序设计语言。

虽说JavaScript已被逐渐应用于浏览器之外的场合，但就目前而言，其主战场还是浏览器。本书除第6部分之外，原则上将JavaScript作为在浏览器中运行的客户端语言。

1.2 JavaScript的历史

JavaScript于1995年登场，运用在当时最流行的浏览器Netscape Navigator中。在此之前，浏览器只能处理HTML与图片，而JavaScript使得浏览器端的程序运行成为可能。

能够在浏览器中运行程序，并非JavaScript的专利。其先驱是另一门著名的程序设计语言Java，主要用于服务器端。当初被称为Java Applet的程序由于可以在浏览器（HotJava）中运行而广受瞩目。

众所周知，尽管Java和JavaScript在保留字和关键字等表层范畴上很相似，但作为程序设计语言，它们之间其实并没有什么关系。JavaScript开发得较晚，开发之初的名称是LiveScript，之后才决定效仿已经颇为有名的Java，改为JavaScript。虽然Java和JavaScript的命名导致了许多误解，但回顾历史，可以说这是一种正确的营销手段。

稍微了解一下语言规则就会发现，Java和JavaScript的执行方式并不像其表面那样相似。JavaScript反而和Ruby或Python这样的轻型脚本语言，或Lisp之类的以函数作为主体的程序设计语言更为相似。不过由于早期主要是跟随Java发展，因此JavaScript的对象名以及方法名和Java比较相似。

JavaScript 简史

在此，我们总结一下 JavaScript 标准的制定时间和一些重要事件（表 1.1）。ECMAScript 将在下一节中进行说明。

表 1.1　JavaScript 简史

年份	事件
1995 年	网景公司开发了 JavaScript
1996 年	微软发布了和 JavaScript 兼容的 JScript
1997 年	ECMAScript 第 1 版（ECMA-262）
1998 年	ECMAScript 第 2 版
1998 年	DOM Level1 的制定
1998 年	新型语言 DHTML 登场
1999 年	ECMAScript 第 3 版
2000 年	DOM Level2 的制定
2002 年	ISO/ IEC 16262:2002 的确立
2004 年	DOM Level3 的制定
2005 年	新型语言 AJAX 登场
2009 年	ECMAScript 第 5 版
2009 年	新型语言 HTML5 登场

最初，JavaScript 所获得的评价并不都是正面的。当时的 PC 性能很弱，JavaScript 的实现也不够成熟，很多人觉得运行了 JavaScript 的页面会变得十分缓慢，浏览器也会变得不稳定。甚至曾经有不少人大力呼吁，应该在浏览器中取消 JavaScript。

随着 Web 使用的普及，要求改善浏览器用户界面的呼声越来越高。因此尽管速度不快，JavaScript 的重要性还是在逐步提升。在这段时期，网景公司以及微软都在不断地进行技术革新，微软逐渐取得技术上的领先地位。由微软等公司提出的 DHTML（动态 HTML）是 JavaScript 的基础。DHTML 是一种为了推广而命名的方便说法，意指 DOM 和 CSS 等 W3C 标准与 JavaScript 相结合后，所能提供的丰富的浏览器用户界面。

就这样，在 2000 年前后，JavaScript 相关的各种技术基本准备就绪。2005 年前后，Web 应用得到广泛普及。特别是出现了以谷歌为首提出的异步 JavaScript（之后统称为 AJAX，即 Asynchronous JavaScript and XML），使接近桌面应用的复杂用户界面得以实现。

在 Web 应用变得越来越复杂的过程中，JavaScript 的代码规模与复杂性也日益提升，prototype.js、jQuery 等各种 JavaScript 库相应登场。可以说，2005 年之后的几年是 JavaScript 的繁荣期。

在这一繁荣期中，还有另一个不能忽视的成员，即 Mozilla 基金会（Mozilla Foundation）。Mozilla 基金会的历史可以追随到网景公司时期。Mozilla 的发展历程不在本书的讲解范畴之内，在此略去，但是 Mozilla 的开源浏览器 Firefox 的坚实发展所带来的 JavaScript 的速度改善，确实是 JavaScript 繁荣的一大主要原因。说到 JavaScript 的性能提升，谷歌在 2008 年与浏览器 Google Chrome 一同发布的 JavaScript 引擎 v8 也是一个重要的契机。在此之后，发生了各种 JavaScript 实现方式之间比拼速度的状况。

1.3　ECMAScript

1.3.1　JavaScript 的标准化

上节提到，JavaScript 是由网景公司提出的。之后，微软开发了和 JavaScript 相兼容的 JScript 并将其应用于 Internet Explorer 中。不过，人们通常将两者统称为 JavaScript。

为了防止因两家公司独自开发而导致 JavaScript 分裂以及其他一些问题，网景公司提出了名为 Ecma

International 的 JavaScript 标准化组织。这一标准语言的名称就是 ECMAScript。由于将语言规则的制定权交给了中立的标准化组织，网景公司放弃了对 JavaScript 的垄断地位，JavaScript 因此具备了标准化程序设计语言所必须的安定感。对于开发者来说，标准的程序设计语言不会随特定企业的想法而轻易改变，也更令人安心。这是因为如果一种语言由某一企业所控制，可能会发生开发终止或是需要收费使用的情况。

ECMAScript 的标准编号是 ECMA-262，并在之后获得了 ISO 的承认（ISO-16262）。通俗来讲，就是得到了 ISO 的权威认证。根据 ECMAScript 标准，网景公司的 JavaScript 被重新定义为一种符合 ECMAScript 标准的程序设计语言。微软的 JScript 亦然。即使之后 JavaScript 的开发主体由网景公司变为了 Mozilla 基金会，这一定义也没有改变。

之后还出现了其他 ECMAScript 的具体实现，不过现在都将它们统称为 JavaScript 实现。严格来说，由网景公司开发、现由 Mozilla 基金会继续发展的语言称为 JavaScript，其他 ECMAScript 标准的实现方式称为 JavaScript 的兼容实现方式。不过这样区分的意义并不大，所以本书将这些统称为 JavaScript 实现方式。目前，具代表性的 JavaScript 实现方式一方面以标准为主，一方面也在独立发展。也就是说，它们在提供了 ECMAScript 功能的基础上，继续提供其他便捷功能。事实上，JavaScript 的具体实现大部分都是 ECMAScript 的超集。因此，如果要保证可移植性，只要做到在代码中仅使用 ECMAScript 标准所包含的功能即可。

1.3.2 被放弃的 ECMAScript 第 4 版

表 1.1（JavaScript 简史）中并没有 ECMAScript 第 4 版，这是因为 ECMAScript 第 4 版没能符合要求而最终被放弃了。

ECMAScript 第 3 版是在 1999 年提出的。一方面可以说 JavaScript 在 10 年间保持了稳定不变，但另一方面也意味着它的标准止于 10 年之前，已经停止了前进。一般来说，1999 以后的 ECMAScript 第 3 版以及 JavaScript 1.5 版被作为默认标准，即使 JavaScript 增加了新功能也被视为增强功能。官方的意见是为了与标准相兼容，不应该使用新功能。标准化有积极的一面，但同时又由于其发展过于缓慢，导致了 JavaScript 的具体实现往往增加了很多独有功能，造成了代码可移植性降低的不良后果。

在大约 10 年的时间里（1999 年至 2008 年），ECMAScript 第 4 版的制定工作一直在进行，原本计划向业已规范有序的标准中进一步加入大量增强功能。在第 4 版中甚至有引入“类”的概念这样大胆的标准变更计划。然而，2008 年的标准化工作大会放弃了大幅度变更标准的计划，转为在第 3 版的基础上进行渐进式改进。于是，在 2009 年直接发布了和第 3 版标准差异不大的第 5 版。

由于 ECMAScript 第 5 版的保守，JavaScript 1.6 版中很多新增功能的处境也变得微妙起来。虽然其中也有一些功能仍然被 ECMAScript 第 5 版采用，但其大部分都没能被接受。因此，虽说只要遵循 ECMAScript 标准依然可以随意使用，但 JavaScript 1.6 版实际上成为了一种独立的 JavaScript 增强版本。总之，如果要遵循标准或是保证可移植性的话，就不应该使用那些功能。

1.4 JavaScript 的版本

正如上一节所讲，JavaScript 是一种符合 ECMAScript 标准的程序设计语言。而事实上，往往是先由 JavaScript 实现某一功能，ECMAScript 才对其进行标准化处理。由于历史原因，Mozilla 基金会所开发的 JavaScript（严格意义上的真正的 JavaScript）常常会在标准化之前就加入一些新功能。

JavaScript 版本和 ECMAScript 版本的对应关系如表 1.2 所示。

表 1.2　JavaScript 的版本

JavaScript 版本	最早采用该版本的浏览器版本	ECMAScript 标准
1.0	Navigator 2.0	–
1.1	Navigator 3.0	以此为基础开始了 ECMAScript 的标准化
1.2	Navigator 4.0–4.05	大致相当于第 1 版标准
1.3	Navigator 4.06–4.7x	第 1 版标准
1.4	–	第 1 版标准
1.5	Navigator 6.0，Mozilla	第 3 版标准
1.6	Firefox 1.5	相当于 ECMAScript 第 4 版的先行版
1.7	Firefox 2	相当于 ECMAScript 第 4 版的先行版
1.8	Firefox 3	相当于 ECMAScript 第 4 版的先行
1.8.1	Firefox 3.5	大致相当于第 5 版标准
1.8.5	Firefox 4.0	第 5 版标准

1.5 JavaScript 实现方式

表 1.3 列出了搭载了 JavaScript 引擎的具有代表性的浏览器。虽说这几年给每个版本附上一个开发代号的做法很流行，不过在这里还是使用各自的通称。

表 1.3　浏览器和 JavaScript 实现方式

浏览器	JavaScript 实现方式
FireFox	SpiderMonkey
Internet Explorer	JScript
Safari	JavaScriptCore
Chrome	v8
Carakan	Carakan（最新版的开发代号）

客户端 JavaScript 代码的可移植性

JavaScript 编程中有一个很麻烦的问题，即在不同的浏览器中其执行方式会有所不同。1.2 节中曾提到 JavaScript 早期的评价并不太好，其中一个很重要的原因就是，JavaScript 在不同的浏览器中的执行方式的确会有差别。许多开发者怨声不断，逐渐造成了一种 JavaScript 编程非常麻烦的印象。但如果冷静下来思考一下，就会发现 JavaScript 其实并没有所说的那么夸张。

稍加了解就会发现，C/C++ 等其他一些语言，和如今的 JavaScript 一样，都衍生出了多种不同的实现方式[①]。它们虽然在遵循语言标准时，能够实现一定程度的可移植性，但对于不同平台（OS）的情况，其可移植性完全无法令人满意。PHP、Perl、Python、Ruby 等流行的脚本语言虽然在不同平台间也有着很高的可移植性，但这是因为它们基本上只有唯一一种实现方式。Java 确实有多种实现方式，也实现了很强的可移植性，不过这是由于它最初就在保证可移植性上花费了很大的精力，所以算是一个例外。把 JavaScript 和 Java 作对比来得出其可移植性不强未免有些不妥。

影响客户端 JavaScript 可移植性的原因主要有两点。

- JavaScript 语言实现方式的不同
- 渲染引擎的差别（DOM 或是 CSS 的解释不同）

在实际中，后者更为麻烦，并由此产生了许多不良开发方式。要解决 JavaScript 语言实现方式差异的关键在于 ECMAScript，因为 ECMAScript 作为一种标准，有明确的规定。现在大多数有名的 JavaScript

① 不过，C++ 的支持者们持有不同意见。由于本书只关注 JavaScript，所以对此不做深究。

实现都基于 ECMAScript 标准，所以只要书写符合 ECMAScript 标准的代码，就能够在很大程度上提高可移植性。

另一方面，渲染引擎没有像程序设计语言一样被标准化，所以相当难办。不过有一个被称为 Acid 的测试，可以用于减少这一不同引擎之间执行方式不同的问题。

http://www.webstardards.org/action/acid3/

Acid 并不像 ECMAScript 那样有明确标准，它会对浏览器进行特定测试，根据返回的结果是否相同来判断代码的执行情况。该测试可以用于判断 JavaScript、DOM、CSS 等各种客户端 JavaScript 的执行情况。现在很多的浏览器都以符合 Acid 标准（即可以通过 Acid 测试）为目标。在执笔本书时，Acid 的版本号为 3。用浏览器登录下面的 URL 地址就能够获得测试得分：

http://acid3.acidtests.org/

刚刚已经介绍了有关客户端 JavaScript 可移植性改进的内容。很可惜，情况尚不乐观。首先是浏览器版本陈旧的问题。之前提到的 ECMAScript 标准以及 Acid 测试标准都是基于最新版本的浏览器的。如果需要支持旧版本的浏览器，则仍然要注意执行方式上的差异。

另一个问题是对 PC 之外的设备的支持。如今的智能手机、平板电脑以及智能电视，原本就有着不同的用户界面。虽说客户端 JavaScript 的可移植性确实在逐渐提高，但如果考虑到现在越来越普及的非 PC 设备的情况，可以说现在正处于一种过去未曾有过的混乱状态。所幸非 PC 设备的渲染引擎基本上被 WebKit 所垄断，总算使问题稍有缓解。

1.6 JavaScript 运行环境

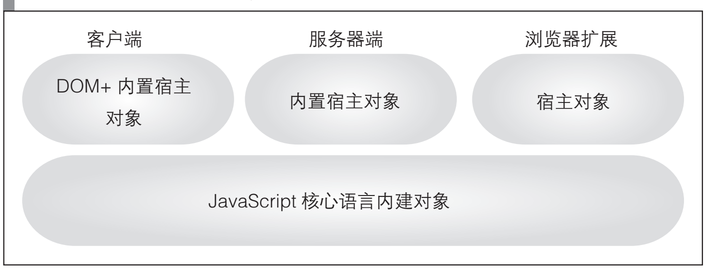
1.6.1 核心语言

由于人们对 JavaScript 的印象大多都是客户端 JavaScript，所以常认为 JavaScript 编程和 DOM 编程是不可分割的。

简单说来，DOM 编程就是浏览器和用户之间的接口，可以在浏览器上显示内容或是反馈用户的点击操作。本书第 3 部分将会对此做进一步详述。尽管在浏览器上两者的联系紧密，但 JavaScript 和 DOM 并不是不可分割的，它们的语言标准相互独立。DOM 对客户端 JavaScript 来说，仅仅是一宿主对象。大家对宿主对象一词可能并不熟悉，只要把它理解为类似于其他程序设计语言的外部库的概念即可，也就是语言中可以更换的部分。而核心语言则是特指 JavaScript 中不可被替代的功能。

JavaScript 的核心语言和宿主对象的概念如下图所示（图 1.1）。

图 1.1 Web 应用程序的组成结构



1.6.2 宿主对象

如图 1.1 所示，JavaScript 中对于不同的运行环境，有着不同的内置宿主对象。这是由于 JavaScript 是被作为一种扩展语言而设计的。对于通用程序设计语言，开发者必须自己开发运行时的上下文环境。正因如此，那些语言才有了通用程序设计语言的名称。另一方面，扩展语言是在内建对象的应用程序（宿主环境）中运行程序的。宿主应用程序会在这时收到一些运行时的上下文信息。JavaScript 会以全局对象作为根节点的对象树的形式，接受这些上下文信息。在启动时，JavaScript 从宿主环境获取的对象树就被称为宿主对象。

从 JavaScript 代码的角度看来，全局对象在程序启动前就已经存在了。客户端 JavaScript 的全局对象被称作 window 对象。

1.7 JavaScript 相关环境

1.7.1 库

大约从 2005 年起，才正式开始使用开源的 JavaScript 库。首当其冲的是 prototype.js。虽然之前也有一些 JavaScript 库，不过直到从 prototype.js 开始，使用库才成为了一种常规做法。prototype.js 受到瞩目的理由之一是它支持多平台 AJAX 处理。

虽然这一时期 AJAX 正在逐步普及，但同时 AJAX 编程也存在重大问题。Internet Explorer 和 Firefox 这两个当时流行的浏览器的 API 并不兼容。而 prototype.js 提供的 API 则是弥合了两者 API 的不同，以解决这一问题。此外，它还提供了很多方便的 API，其中一些可以用于扩展 DOM 的功能，另一些则是 Ruby 迭代器的衍生。

在此之后，如表 1.4 所示，出现了大量的 JavaScript 库。

表 1.4 代表性的客户端 JavaScript 库

名称	发布 URL
prototype.js	http://www.prototypejs.org/
script.aculo.us	http://script.aculo.us/
jQuery	http://jquery.com/
Ext.js	http://www.sencha.com/products/extjs/
Yahoo! UI Library(YUI)	http://developer.yahoo.com/yui/
Dojo	http://dojotoolkit.org/
MochiKit	http://mochi.github.com/mochikit/
MooTools	http://code.google.com/closure/libray/
uupaa.js	http://code.google.com/p/uupaa-js/

1.7.2 源代码压缩

为了使客户端 JavaScript 的执行更加高速，可以对源代码进行压缩。通过压缩源代码可以实现以下效果以提高执行速度。

- 减少了网络通信传送量而使得网络等待时间减少。
- 源代码缩短之后，JavaScript 解释器（浏览器）用于解释代码的时间减少。
- （有些压缩工具可以使）源代码得到优化。

代表性的源代码压缩工具如表 1.5 所示。

表 1.5 源代码压缩工具

名称	URL
Google Closure Compiler	http://code.google.com/closure/compiler/
YUI Compressor	http://developer.yahoo.com/yui/compressor/
packer	http://dean.edwards.name/packer/
JSMin	http://www.crockford.com/javascript/jsmin.html

单纯的压缩工具的效果只是删除不需要的空白内容、换行符以及注释等。为了提高运行速度而不写必要的注释并不是可取的做法，所以，这样单纯的压缩工具也是有其存在意义的。稍高级一些的压缩工具则会进行将变量名替换为较短的字符串之类的处理，不过这样一来，源代码的可读性也会大大降低。更高级一些的压缩工具能够像大多数的编译器那样对代码进行优化。例如，去除无用的代码，或是预先计算代码中的一些表达式，并将其替换为常量，等等。而要实现这一效果，就不能把源代码仅看作是单纯的字符串，还要以 JavaScript 的标准正确地解释其含义。这样一来，也就实现了对代码的检查，能够发现代码中一些潜在的错误。

虽然对源代码进行压缩非常地麻烦，但相应的也能获得不小的收获。因而，在开发规模较大的情况下，应当对源代码进行压缩。

1.7.3 集成开发环境

JavaScript 已经有了不少的集成开发环境（IDE），不过其中的一些还尚未完善。代表性的 IDE 如表 1.6 所示。

表 1.6 JavaScript 的 IDE

名称	说明
Orion	Eclipse 基金会提供的基于 Web 的 JavaScript 专用 IDE
Cloud9	云端在线（http://cloud9ide.com）JavaScript 专用 IDE
Eclipse	Eclipse 基金会提供的 IDE。作为 Java IDE 而闻名，同时也支持 JavaScript
NetBeans	由 Oracle（原 Sun）开发。作为 Java IDE 而闻名，同时也支持 JavaScript
Aptana Studio	Appcelerator（Titanium 的开发商）所收购的 Aptana 公司（http://www.aptana.org）的免费产品
WebStorm	JetBrains 公司的收费产品（http://www.jetbrains.com/webstorm/）
Komodo IDE	ActiveState 公司的收费产品



第 2 部分

JavaScript 的语言基础

这一部分将讲解 JavaScript 的核心概念。其中，不仅会介绍 JavaScript 的语法规则，还将介绍符合 JavaScript 风格的代码书写方式，以及 JavaScript 的编程理念。

第 2 章 JavaScript 基础

在详述 JavaScript 的语言基础之前，我们先做个简单介绍。本章的目的是使大家对 JavaScript 语言基础有整体性的把握，而严谨详细的说明将留在之后的章节中详述。

2.1 JavaScript 的特点

JavaScript 程序设计语言有如下几个特点：

- 解释型语言
- 类似于 C 和 Java 的语法结构
- 动态语言
- 基于原型的面向对象
- 字面量的表现能力
- 函数式编程

■ 解释型语言

JavaScript 是一种解释型语言，和解释型语言相对的是编译型语言。解释型语言直接在运行环境中执行代码，所以一般来说，与编译型语言相比，解释型语言的开发更为容易。特别是 JavaScript，其运行环境是已经普及的浏览器，所以能够很容易地尝试开发。这是其他程序设计语言所不能比拟的。

解释型语言的劣势在于，其运行速度通常都会慢于编译型语言，不过这也只是理论上的情况。现在，解释型语言和编译型语言之间的界线正在变得越来越模糊。编译型语言在有了足够快速的编译器和功能强大的开发环境之后，也能实现和解释型语言相匹敌的开发难易度。同时，解释型语言由于使用了 JIT（Just In Time）这种能够在运行中进行编译的技术，使得运行速度得以改善。

如今，在选择程序设计语言时，比起选择编译型语言还是解释型语言，更重要的是考虑语言的设计目的。是为了使开发过程变得轻松还是为了提高执行效率，语言最初的设计理念不同，其性质自然会有差异。设计 JavaScript 之初，优先考虑的是使开发过程变得轻松，因此提供了多种特性。

■ 类似于 C 和 Java 的语法结构

JavaScript 的语法结构与 C 和 Java 相似。JavaScript 同样有 if 或 while 这类关键字，其语法结也与 C 和 Java 类似。它们乍一看很像，因此有这些语言开发经验的人很容易就能熟悉 JavaScript。不过需要注意的是，它们之间的相似性其实并不如表面看起来的那么强。

■ 动态语言

JavaScript 与 C 和 Java 所不同的一点在于，JavaScript 是一种动态语言，将在之后详述。单从代码的角度看，动态语言的变量和函数是不指定返回值类型的。JavaScript 之所以被设计成动态语言，和选择将其设计为解释型语言的理由一样，都是优先考虑了开发难易度的结果。对解释型语言以及动态语言的特性的喜好虽然见仁见智，但语言本身并没有高下优劣之分。

■ 基于原型的面向对象

解释型动态语言并不少见，现有的较为知名的脚本语言大多都属于这一类型。不过基于原型的面向对象特性，使得 JavaScript 与它们有所不同。基于原型的面向对象特性和基于类的面向对象特性是有所差别的，在此请先了解这一点即可，更为详细的内容将会在之后详述。目前，被称为面向对象语言的程序设计语言，大多提供了基于类的面向对象语言功能。JavaScript 虽然并不是第一个采用基于原型的面向对象特性的语言，不过可以说是这类语言中最为著名的。同样，基于原型与基于类的面向对象语言之间的差异，也主要是个人喜好的区别，而并非是孰优孰劣的问题。

■ 字面量的表现能力

字面量的表现能力是 JavaScript 开发生产力得以提高的一个重要原因。在 Perl 之后，很多语言都提供了功能强大的字面量功能。虽然其中表现突出的不止 JavaScript 一种，不过由于它的字面量功能相对来说非常优秀，所以作为语言特点之一列举于此。

■ 函数式编程

最后来介绍一下函数式编程。函数式编程是一种历史悠久，而又在最近颇为热门的话题。函数式编程在面向对象一词诞生以前就已经存在，不过它在很长的一段时间里都被隐藏于过程式编程（面向对象也是过程式编程的一种）的概念之下。然而现在这种状况正在逐步发生改变，JavaScript 正是这一改变过程中的一部分。尽管 JavaScript 能直接支持的程序设计范式在本质上还是过程式的，但由于具备了匿名函数，可以把函数作为对象来使用，所以同时也能够支持函数式编程。

2.2 关于编排格式

在这一部分中，我们将按照以下方式表示 JavaScript 的代码范例。原则上使用 smjs（SpiderMonkey 的 Shell）来确认执行结果，并使用以 ECMAScript 第 5 版为标准的 JavaScript 1.8.5。在需要明确说明执行结果时，在 JavaScript 代码之后附有相应的执行结果。

表达式求值的结果也采用和执行结果相同的方式编排。

```
js> var s = 'foorbar';        // 用于说明语句含义的注释
运行结果（或是表达式求值的结果）
```

在 JavaScript 中用于分隔语句的分号是可以省略的，不过在本书的范例代码中，不会省略分号。

如果在运行 smjs 时发生错误，会像下面这样在行首显示“typein: 数字”。这里的数字表示正在运行的代码行号。由于运行环境不同，本书将省略“typein: 数字”这样的错误行号。

```
js> x;
typein:1 ReferenceError: x is not defined
```

print 函数

JavaScript 的核心语言中并不包括 print 函数，但是在本书的代码范例中，将会使用到 print。如果要在浏览器中运行相关代码，请改用 alert 或是 document.write ；如果使用 FireBug 或是 Node.js（请参见本书第 6 部分），请改用 console.log 函数。

```
// 浏览器
var print = alert;
或者
var print = document.write;
```

```
// 使用 FireBug 或是 Node.js 时
var print = console.log;
```

2.3 变量的基础

2.3.1 变量的使用方法

本节将对 JavaScript 中变量的使用方法进行说明。变量的作用是给某一个值或是对象标注名称。在介绍了对象和函数之后，会再对变量这一主题进行详细说明。本节中先不考虑那些复杂情况，而着重说明变量的使用方法。

像下面这样使用关键字 var 就可以对变量进行声明。

```
js> var foo;                // 声明变量 foo
```

在之后还会进一步详述变量名中具体可以使用哪些字符，现阶段请先将其理解为变量名可以使用任意的英文字母即可。通过赋值运算符（=）可以给变量赋值，即在运算符的左侧书写变量，而在其右侧书写要赋的值。

```
js> foo = "abc";            // 将字符串 "abc" 赋值给变量 foo
```

变量的声明和赋值也可以像下面这样同时进行。在声明变量的同时为其赋值是一种较好的编程风格。

```
js> var foo = "abc";        // 声明一个赋值为字符串 "abc" 的变量 foo
```

JavaScript 中的变量没有变量类型（之后会说明变量类型的概念）。因为没有变量类型，所以对于同一个变量，既可以赋值为字符串，也可以赋值为数字，就像下面这样。不过通常情况下，以这种方式来使用变量并不是好习惯，所以请尽可能避免出现这样的代码。

```
js> var foo;
js> foo = "abc";            // 将字符串 "abc" 赋值给变量 foo
js> foo = 123;              // 将数值 123 赋值给变量 foo
```

在表达式中写上某个变量名之后就能获取该变量的值。

```
js> var n = 7;              // 将数值 7 赋值给变量 n
js> n + 1;                  // 获取变量 n 的值，并加上 1
8
```

严格来说，语句中的变量，对于左值和右值是有所不同的。左值指的是赋值表达式 = 左侧的变量名，右值指的是赋值表达式 = 右侧或是在赋值表达式外的其他表达式中出现的变量名。右值中的变量是所要用于赋值的值，而左值中的变量则是将被赋值的对象。用这些专业术语来说明可能不太容易理解，其实在这一点上，JavaScript 和大部分支持赋值功能的程序设计语言是完全相同的。对于左值和右值，只要根据直觉，将其理解为被赋值的对象与所赋值的来源即可。

被声明但未进行任何赋值的变量，其值为 undefined（之后会说明 undefined 值的含义）。读取这类变量的值不会引起运行时错误。需要注意的是，在大部分情况下，读取 undefined 值都是产生错误的根源。

```
js> var foo;
js> print(foo);             // 变量 foo 的值为 undefined
undefined
```

如果要读取没有被声明的变量（即作为右值使用该变量），就会引发 ReferenceError 异常；如果将其作为左值使用，即作为赋值对象使用，则不会发生错误。

更为详细的内容请参见 2.3.2 节 。

```
js> print(x);
ReferenceError: x is not defined
```

2.3.2　省略 var

熟悉 JavaScript 的人也许知道，在 JavaScript 中 var 关键字是可以省略的。尽管此前提到，变量是通过 var 来声明的，但其实不通过 var 来声明也可以对变量进行赋值。这样的变量称为隐式声明变量。采用隐式声明的变量都是全局变量，即使是在函数内部隐式声明的变量也属于全局变量。

在函数外部通过 var 声明的变量也是全局变量，这类全局变量是显式声明的。为了和显式声明的全局变量相区别，那些没有通过 var 声明的变量被称为隐式全局变量。

应当尽可能避免使用全局变量，特别是应该避免使用隐式全局变量。开发者只需做恰当的处理，即在声明变量时总是使用 var，就可以完全避免使用隐式全局变量，从而解决这一问题。

不仅是本书，只要是稍微专业一些的 JavaScript 书籍，都不会推荐使用隐式全局变量。而且在 ECMAScript 第 5 版的 strict mode 中，隐式全局变量已经被判定为一种错误（请参见第 2 部分第 7 章的“专栏”），所以请不要省略 var。

2.3.3　常量

ECMAScript 标准没有规定常量的声明语法。不过在 JavaScript 的自定义增强功能中，是可以对常量进行声明的。由于是自定义的增强功能，因此并没有明确的规范。下面介绍的是 SpiderMonkey 的情况。

如果要声明一个常量，需要使用 const 关键字而不是 var。可以作为常量名使用的字符和变量的是相同的，不过习惯上常量名都以大写字母表示。const 的使用方法如下所示。

```
js> const FOO = 7;    // 声明常量
js> print(FOO);
7
```

即使给常量再次赋值，这个常量的值也不会发生改变。其实，对常量再次赋值应该算作一种错误，但在实际中这并不会导致出错，对此请多加注意。

```
js> const FOO = 7;
js> FOO = 8;          // 对常量再次赋值
js> print(FOO);       // 常量的值不会发生改变
7
```

如果在声明时没有对常量进行赋值的话，它的值就是 undefined，对其的处理方式和变量相同。由于 const 属于自定义增强功能，所以在本书中不会对此深究。

```
js> const FOO;
js> print(FOO);
undefined
```

2.4　函数基础

2.4.1　函数的定义

JavaScript 中的函数是一种类似于 Java 中方法的语言功能，不过它可以独立于类进行定义，所以从表面上来看，反而和 C 语言或是 PHP 的函数、Perl 的子程序更为相似。不过，JavaScript 中的函数和它们在本质上是不同的。它们之间的差异以及其他详细的内容，将在第 6 章中再具体说明。

本节仅对 JavaScript 的函数进行概要性的说明。在基本使用方式上，JavaScript 中的函数和其他程序设计语言中被称为函数或子程序的概念并没有什么不同。函数是由一连串的子程序（语句的集合）所组成的，可以被外部程序调用。向函数传递参数之后，函数可以返回一定的值。

通常情况下，JavaScript 代码是自上而下执行的，不过函数体内部的代码则不是这样。如果只是对函数进行了声明，其中的代码并不会执行。只有在调用函数时才会执行函数体内部的代码（代码清单 2.1）。

代码清单 2.1　包含函数的代码的执行顺序

```
print('1');
function f() {    // 声明函数
    print('2');
}
print('3');
f();              // 调用函数
```

```
// 代码清单 2.1 的运行结果
1
3
2
```

2.4.2　函数的声明与调用

可以通过函数声明语句来定义一个函数。函数声明语句以关键字 function 开始，其后跟有函数名、参数列表和函数体。其语法如下所示：

```
// 函数声明语句的语法
function 函数名 ( 参数 , 参数 , ……) {
  函数体
}
```

代码清单 2.2 是个具体例子，其中函数名为 sum，参数名为 *a* 和 *b*。函数声明中所写的参数称为形参（形式参数）。代码清单 2.2 中的函数 sum 对两个参数做了加法运算，并通过 return 语句返回结果。

代码清单 2.2　函数 sum 的声明

```
function sum (a, b) {
    return Number(a) + Number(b);
}
```

可以像下面这样来调用函数 sum。调用函数时，传递给函数的参数称为实参（实际参数）。下面代码中以 3 和 4 作为实参调用了函数 sum。

```
// 函数 sum 的调用
js> sum (3, 4);
7
```

函数声明时不必指定形参的类型[①]。任何类型的值都可以作为实参传递，因而开发者在设计函数时需要考虑接收错误类型的值的情况。此外，形参的数量和实参的数量可以不一致，这一点将在之后再具体说明。JavaScript 的这些特性，与始终严格检查参数类型的 Java 形成了鲜明的对比。因此，在 JavaScript 中自然也就不存在函数重载这一特性（即可以存在多个参数不同的同名函数）。

2.4.3　匿名函数

还可以通过匿名函数表达式来定义一个函数。其语法形式为在 function 后跟可以省略的函数名、参

① 不过 JavaScript 的变量本身就没有类型可言，所以形参没有类型也不奇怪。

数列表以及函数体。其语法如下所示：

```
// 匿名函数的语法
function （参数，参数，……）{
  函数体
}
function 函数名 （参数，参数，……）{
  函数体
}
```

可以看到，函数声明语句和匿名函数表达式在语法上几乎一模一样，唯一的区别仅仅是能否省略函数名称而已。不过，因为匿名函数表达式是一种表达式而非语句，所以也可以在表达式内使用。另外由于它是表达式因此也会有返回值。匿名函数的返回值是一个 Function 对象的引用（关于引用的详细内容将在第 5 章中进行说明）。把它简单理解为返回一个函数也没有问题。

不过请不要因为它们都是表达式，而将匿名函数表达式与函数调用表达式相混淆。函数调用表达式在大部分程序设计语言中都是存在的，而匿名函数表达式在一些程序设计语言中并不存在（至少 Java 中的方法是无法实现这样的功能的）。其实，通过表达式来定义一个函数并不是什么新的功能。早在与 JavaScript 有些类似的 Lisp 语言的时代，这种功能就已经存在，并且在一些比较新的程序设计语言中，这一功能正在变得越来越常见。

匿名函数表达式的使用方式如代码清单 2.3 所示。赋值表达式右侧的就是匿名函数表达式。

代码清单 2.3　匿名函数表达式的例子

```
var sum2 = function (a, b) {
    return Number(a) + Number(b)
}
```

sum2 的前面是 var，所以它是一个变量名。以 function 开始的匿名函数表达式将返回一个函数。也就是说，代码清单 2.3 的含义是，将 Function 对象的一个引用赋值给变量 sum2。可以像下面这样来调用变量 sum2 所引用的函数。

```
// 调用函数 sum2
js> sum2(3, 4);
7
```

这段代码和代码清单 2.2 中调用函数 sum 的方式没有区别。也就是说，代码清单 2.2 中的函数声明语句，和代码清单 2.3 中赋值表达式的作用是相同的，都会将匿名函数表达式赋值给变量。两者都是在生成一个没有名称的函数体（Function 对象）之后，再赋予其一个名称。目前只要认为代码清单 2.2 和 2.3 中的语句是相同的就可以了。第 6 章会对两者的细微差异进行说明。

还可以像下面这样，在右侧书写通过函数声明语句进行定义的函数，以将其赋值给左值，这样就可以通过被赋值对象的名称来调用该函数。将其理解为 sum 这一名称持有该函数对象的引用即可。至此，读者或许会觉得变量名和函数名之间的分界很模糊，而事实也确实如此，我们之后将会对此进行详述。

```
// 在表达式右侧书写代码清单 2.2 中的函数名
js> var sum3 = sum;
// 调用函数 sum3
js> sum3(3, 4);
7
```

2.4.4　函数是一种对象

JavaScript 中的函数和 Java 中的方法或 C 语言中的函数的最大不同在于，JavaScript 中的函数也是一种对象。下一节将阐述对象的概念，届时可以了解到对象在本质上是没有名称的。而对于函数来说也是

如此，因为函数本身也是一种对象。

正如变量存在的意义是为了调用没有名称的对象，函数名存在的意义是为了调用没有名称的函数。因此，变量名和函数名实质上是相同的。这一点在之后会再次说明。虽然有时也需要区别对待变量名和函数名，不过这里为方便起见，暂且认为两者在本质上是相同的。

JavaScript 的函数是一种对象，不过并不是说所有的对象都是函数。函数是一种包含了可执行代码，并能够被其他代码调用的特殊的对象。

2.5 对象的基础

2.5.1 对象的定义

从底层实现来看，JavaScript 的对象和 Java 的对象在基本原则上是相同的。两者都是内存中的实体，保持着某种状态，并且是用于编程操作的目标对象。但是，从高层概念来看的话，就会发现两者有着不小的差别。

Java 中的对象可以认为是类的一种实例化结果，而 JavaScript 中并没有类这样的语言构造。JavaScript 中的对象是一个名称与值配对的集合。这种名称与值的配对被称为属性。这样一来，JavaScript 对象可以定义为属性的集合。

表面上看，JavaScript 对象和 Java 的映射（java.util.Map）非常相似。实际上，JavaScript 对象可以用作管理键值对的关联数组 [又称映射（Map）或字典（Dictionary）]。JavaScript 对象还有着 Java 映射所没有的两个特点。

其一是 JavaScript 对象的属性值可以由函数指定。

其二是 JavaScript 具备一种称为原型链的构造。通过这一构造，JavaScript 对象实现了类似于类的继承的能力。具体的内容将在第 5 章中进行详细说明。

以上的说法可能有些复杂，简单说来，将对象理解为一种实体即可，程序可以通过它来进行数据处理[①]。

2.5.2 对象字面量表达式与对象的使用

可以通过对象字面量表达式来生成一个对象。对象字面量表达式由大括号 {} 括起，内部有属性名和属性值，如下所示。

```
// 对象字面量表达式的语法
{ 属性名 : 属性值 , 属性名 : 属性值 , …… }
```

属性名可以是标识符、字符串值或是数值。属性值则可以是任意的值或对象。具体例子如代码清单 2.4 所示。

代码清单 2.4　对象字面量表达式的例子

```
{ x: 2, y:1 }                                            // 属性名是标识符
{ "x":2, "y":1 }                                         // 属性名是字符串值
{ 'x':2, 'y':1 }                                         // 属性名是字符串值
{ 1:2, 2:1 }                                             // 属性名是数值
{ x:2, y:1, enable:true, color:{ r:255, g:255, b:255 } } // 各种类型的属性值
```

在 ECMAScript 第 5 版中，还允许下面这样以逗号结尾的对象字面量，而这在 ECMAScript 第 3 版中是被禁止的。由于会在版本较老的 Internet Explorer 中发生问题，所以应当尽可能避免在对象字面量的最

① 更进一步的说明请参见第 5 章。

后以逗号结尾。

```
{ x:2, y:1, }      // ECMAScript 第 5 版将会忽略最后一个逗号，所以不会产生问题
```

对对象字面量表达式求值所得到的结果，是所生成对象的一个引用。

像下面这样，在赋值表达式的右侧书写对象字面量的话，就能够将对象的引用赋值给变量（将会在第 5 章中对引用进行详细说明）。

```
// 对象字面量表达式与赋值表达式
js> var obj = { x:3, y:4 };     // 所生成对象的引用将被赋值给变量 obj
js> typeof obj;                 // 通过 typeof 运算符来判别 obj 的类型，得到的结果是 object
object
```

为方便起见，我们称变量 obj 所引用的对象是对象 obj。5.2.3 节还将对此详细说明。

2.5.3　属性访问

可以通过点运算符（.）访问对象引用中的属性。只要在点运算符之后书写属性名，就能够读取相应的属性值。

```
// js> print(obj.x);        // 显示对象 obj 的属性 x 的值
3
```

如果属性的值是一个对象，可以像下面这样通过多次点运算来读取其属性。

```
js> var obj2 = {pos: { x:3, y:4 } };
js> print(obj2.pos.x);
3
```

在赋值表达式的左侧书写属性访问表达式的话，就可以将相应的值赋给该属性。

```
js> obj.x = 33;             // 这将覆盖已有的属性值
js> print(obj.x);
33
```

如果赋值给尚不存在的属性名，则将新建该属性并对其赋值。

```
js> obj.z = 5;              // 新建属性
js> print(obj.z);
5
```

2.5.4　属性访问（括号方式）

除了点运算符外，还可以使用中括号运算符 [] 来访问属性。[] 内是需要访问的属性名的字符串值。

```
js> print(obj['x']);        // 与 obj.x 相同
3
```

[] 里可以是字符字面量，也可以是值为字符串的变量。

```
js> var name = 'x';
js> print(obj[name]);       // 与 obj.x 相同
33
```

括号运算符也能用于赋值表达式的左侧。

```
js> obj['5'] = 5;           // 将数值 5 赋值给该属性（若该属性不存在则新建属性）
```

第 5 章将对分别在哪些场合下使用点运算符以及括号运算符进行说明。

2.5.5 方法

可以把任意类型的值、对象或者函数赋值给对象的属性。正如前节所讲，对匿名函数表达式求值所得到的结果是函数对象的引用，所以，也可以像下面这样来书写。

```
js> obj.fn = function (a, b) {return Number(a) + Number(b); };
                     // 将函数赋值给对象 obj 的属性 fn
```

可以像下面这样，对被赋值给属性的函数进行调用。

```
js> obj.fn(3, 4);     // 调用函数
7
```

回顾一下之前章节的说明可以发现，在代码清单 2.2 之后还可以像下面这样书写。

```
js> obj.fn2 = sum;    // sum 是在代码清单 2.2 中定义的函数
js> obj.fn(3, 4);     // 调用函数
7
```

从该调用表达式可以看出，通过点运算符来调用函数，和其他语言中的方法调用十分相似。事实上，不仅在表面上很像，其内部原理也和方法调用如出一辙。在 JavaScript 中并没有方法这种语言特性，不过实际上作为对象属性的函数也可称为一种方法[①]。

2.5.6 new 表达式

JavaScript 中 new 表达式的作用是生成一个对象。可以像下面这样使用该表达式。

```
// new 表达式的例子
js> var obj = new Object();
js> typeof obj;       // 通过 typeof 运算符来判别 obj 的类型，得到的结果是 object
object
```

和之前说明的通过对象字面量表达式生成的对象一样，通过 new 表达式生成的对象，其属性能够被读取。

以直观的方式来理解的话，关键词 new 之后所写的是类名。不过正如已此前说明，JavaScript 中没有类的概念，所以，根据 JavaScript 的语法规则，new 之后所写的是函数名。在 new 之后写函数名的话，就会把该函数作为构造函数来进行调用。

2.5.7 类与实例

再次强调一下，在 JavaScript 的语言特性中没有“类”的概念。但是在本书中为了便于理解，将用类这个词来称呼那些可以被视作“类”的概念。也就是说，在本书中将用“类”，来称呼那些实际上将会调用构造函数的 Function 对象。此外，在强调对象是通过调用构造函数而生成的时候，会将这些被生成的对象称作对象实例以示区别。

综上所述，虽然在 JavaScript 中没有类的概念，但将 new 之后所写的标识符（函数名）看作是类名，也并没有什么概念上的问题。也就是说，完全可以认为，上文中代码 new Object() 的作用是生成一个 Object 类的实例。

2.5.8 对类的功能的整理

在这一部分中，将集中对一些标准内置类（Object 类或是 String 类）的功能进行说明。现在将类的

① 在第 5 章 中将会对方法进行详细说明。

功能整理在表 2.1 中。

表 2.1　对类的功能的整理

接口	说明
函数或是构造函数的调用	–
类的属性	相当于 Java 中的 static 方法或是 static 域
prototype 对象的属性	相当于 Java 中的实例方法
实例属性	相当于 Java 中的实例域

可能大家阅读了第 5 章之后，才能彻底理解表 2.1 的含义。在这里，先对此做一些简单的说明。

类的属性是一个类自身的属性，例如，String 类的属性是 String 类的对象自身的属性。如果是函数的话，则可以像 String.fromCharCode(0x41) 这样来使用。如果用更加直观一些的说法来讲，这就相当于 Java 或 C++ 中的 static 方法。

prototype 对象的属性和实例属性，都是以对象实例的形式来进行访问的。以 String 类为例，可以以 str.trim() 或是 str.length 的方式，来使用引用了 String 对象（对象实例）的变量 str。

prototype 对象的属性与实例属性之间的不同点在于是否进行了继承。例如，String 对象的 trim 方法，其实是 String.prototype 对象的属性。这种以实例来继承属性的方式被称为原型继承。

2.5.9　对象与类型

Java 中对象分为类与接口等不同类型。在 JavaScript 中不存在这样的类型区分。不过如果把对象的行为方式定义为其类型的话，JavaScript 的对象也可说是有类型的，只不过并不是 Java 那样的严格的类型。

在 Java 中，需要事先进行严格的类型定义（通常以层级方式进行管理），然后将对象置于类型层级中进行分类。然而在 JavaScript 中，则是根据不同对象的不同行为方式，在事实上对其类型做出了分类。在 Java 中，由于语法规则的需要，强制使用了基于类型层级的编程风格。与此相对，在 JavaScript 中虽然也能够定义（类似于）类（的实体），并以类型层级的方式对对象进行分类，但这仅仅是可供选择的一种风格，没有作强制性的要求。

2.6　数组的基础

数组是一种用于表达有顺序关系的值的集合的语言结构。在 JavaScript 中，数组并非是一种内建类型。相对地，JavaScript 支持 Array 类，所以数组能够以 Array 类的实例的形式实现。不过，由于有数组字面量的表达方式，所以在一般情况下，只需将其作为内建类型使用即可。

数组字面量的书写方式为，在方括号内列出所需的值。通过数组字面量就能够生成数组。

```
// 数组字面量的例子
js> var arr = [1, 100, 7];
```

数组内的各个值被称作元素。每一个元素都可以通过索引（下标）来快速读取。索引是从零开始的整数。对于上面的数组，可以像下面的代码这样，通过在方括号中书写索引值 1 来读取其第 2 个元素。

```
// 接之前的代码
js> print(arr[1]);          // 读取索引值为 1 的元素
100
js> arr[1] = 200;           // 为索引值为 1 的元素赋值
js> print(arr[1]);
200
```

在括号中不仅可以直接写某个数值，还可以写具有某一特定值的变量或表达式。

```
// 接之前的代码
```

```
js> var n = 1;
js> print(arr[n]);         // 与 a[1] 含义相同
200
js> print(arr[n + 1]);     // 与 a[2] 含义相同
7
```

JavaScript 的数组支持同时包含不同类型的元素。以下是一个同时包含数值和字符串的例子。同样，对象以及数列也能够作为数组元素。

```
// 包含不同类型元素的数组的例子
js> var arr = [1, 'foo', 7];
```

专栏

代码书写风格

无论是哪种程序设计语言，都有其代码书写风格。虽然不遵循代码书写风格也能写出可以运行的代码，但是这会为之后阅读代码时增加不必要的麻烦。因此遵循代码书写风格是非常重要的。

JavaScript 由于其特殊的历史与定位，有着略显独特的代码书写风格。造成这一局面的背景之一，和客户端 JavaScript 的历史有关。对于客户端代码来说，代码体积的大小不但会影响运行速度，还会直接影响网络传输的性能。在 Web 的发展历史中，尤其是早期，如何减少网络数据传输量是一个重要的课题（至今仍然非常重要）。因此，JavaScript 中有着大量以减少代码书写量为目的的代码书写风格。乍一看，简洁书写似乎是一件好事，但事实上，也存在着一些看似取巧实则糟糕的做法。说到底，代码书写风格是历史的产物，不能仅仅根据是否正确来判断代码书写风格的好坏，而应该去尝试接受这些既定事实。

另一背景则和 JavaScript 普及的历史有关。其他很多程序设计语言在被广泛普及之前，往往都会有少数优秀的开发者首先使用。通常在这一时期，该语言的代码书写风格会初步成形，然后在随后更为广泛的普及过程中发生一些变化。而 JavaScript 的情况并非如此。在其普及初期，对 JavaScript 感兴趣的主要是那些书写 HTML 的网页设计师，或者一些主用其他语言的开发者，他们仅仅把 JavaScript 作为一种临时的替代品来使用。因此，JavaScript 没有属于自己的核心代码书写风格，却有着很多模仿其他语言而来的代码书写风格。比如，类似于 Java 代码风格的 JavaScript 代码，又或是类似于 PHP 代码风格的 JavaScript 代码，诸如此类。不过最近几年，具有 JavaScript 自身特点的代码书写风格正在逐渐普及。

第 3 章 JavaScript 的数据类型

在 JavaScript 中，除了 Object 类型，还有 5 种基本数据类型。由于 JavaScript 是一种动态语言，因此在使用时很少会意识到其数据类型的存在。但是为了深入理解 JavaScript 语言，我们仍然有必要清楚地理解各种数据类型。

3.1 数据类型的定义

数据类型决定了一个数据的特征，即限定了该数据必须按照一定的规则进行操作。在程序设计中也是如此，特定数据类型的数据会有其相应的行为模式。

JavaScript 中有以下 5 种基本数据类型。

- 字符串型
- 数值型
- 布尔型
- null 型
- undefined 型

在这 5 种基本数据类型之外的都被称为 Object 类型。也就是说，总的来看，JavaScript 中的数据类型可以分为 6 种。

在这里，先来解释我们已经接触过的两个术语，即说明值和对象在使用上的区别。在本书中，基本数据类型的实例被称为“值”，Object 类型的实例被称为“对象”。其他面向对象语言也是这样区分的，所以不会造成误解。

不过，JavaScript 支持值与对象的隐式变换，所以有时可认为两者是完全相同的，而这会导致理解混乱。同时，JavaScript 的变量没有类型之分，所以值和对象之间的区别就变得更为模糊了。

3.1.1 在数据类型方面与 Java 作比较

JavaScript 和 Java 在数据类型方面的差异，不仅仅是两者所采用的术语不同。接下来，我们从两个不同角度来讨论这些差别：其一是从动态数据类型和静态数据类型的角度，其二是从基于类和基于原型的角度。

■ 动态数据类型与静态数据类型

在 JavaScript 中，值和对象具有数据类型，而变量没有数据类型。事实上，JavaScript 中并不存在变量类型的概念。与之相反，在 Java 语言中变量有类型之分。Java 中的变量有其数据类型，它限制了可以对该变量赋值的值或对象引用的类型。因为 JavaScript 的变量不具有数据类型，所以可以对其赋任意类型的值，也可以使其引用任意类型的对象。

像 Java 这样，变量具有数据类型的语言，被称为静态数据类型语言；而像 JavaScript 这样，变量没

有类型的语言，则被称为动态数据类型语言[①]。

■ 基于类与基于原型

对于 Java 来说，内建类型（int 或 double 之类）之外的都是用户自定义类型。用户自定义类型又可以分为类和接口两种类型。Java 的用户自定义类型的使用方法，从其名称中就可略知一二，即开发者需要书写该类型的定义语句来定义该类型。而对象则作为这些由用户定义的数据类型的实例（实体）存在。这就是 Java 的基本特性。这种编程风格被称为基于类的语言风格。

另一方面，在 JavaScript 的语言规范中，不存在定义数据类型的语句。不需要使用特别的语句就能定义一个对象的属性或方法，而这样也就决定了该对象的类型。所谓类型也就是行为方式上的共性。由于每个对象都具有共同的行为方式，所以可以使用原型对象。这样的编程风格被称为基于原型的风格。

3.1.2 基本数据类型和引用类型

虽然 JavaScript 的变量不具有数据类型，但从概念上，JavaScript 变量可以分为基本数据类型变量和引用类型变量。基本数据类型变量直接保存有数值等类型的数据的值，而引用类型变量则保存有对象的引用。尽管表面上两者没有区别，但其内在是不同的。因此为了正确地理解其内部实现原理，就需要引入引用这一概念。

这一部分内容将在第 5 章进一步详细说明。

3.2 内建数据类型概要

在 ECMAScript 标准中，内建数据类型（built-in type）分为 5 种基本数据类型以及 Object 类型。该标准中并没有原始类型（primitive type，也称为基本数据类型或是简单数据类型）这样的术语，而是使用了原始值（primitive value）的名称。在本书中，为了便于理解，统一使用基本数据类型这一名称。

JavaScript 的基本数据类型

这里通过与 Java 的对比来解说 JavaScript 的基本数据类型。

JavaScript 的字符串型是基本数据类型，而 Java 的字符串型并不是基本数据类型，这是两者的区别之一。不过，其实两者在本质上并没有太大的不同。因为在 Java 中，字符串型和字面量以及运算符一样，属于被特别对待的 Object 类型。字符串连接运算符（+ 号）在 Java 和 JavaScript 中的作用也是相同的。之后还将说明，在 JavaScript 中，字符串值会被隐式地转换为字符串对象类型。熟悉 Java 的人很容易就能掌握 JavaScript 中字符串型的用法。不过在 JavaScript 中不存在字符类型。如果需要表达某个字符的话，请使用长度为 1 的字符串值。

JavaScript 只有一种数值类的数据类型，其内部构造为 64 位的浮点小数，这相当于 Java 中的 double 类型。和 JavaScript 的情况不同，在 Java 中有 5 种整数类型和 2 种浮点数类型。JavaScript 之所以只支持一种数值类型，是由于设计当初更多地着眼于降低编程难度，而非提升运行效率。如果有多种数值类型，就不得不在考虑在进行赋值时考虑是否会产生错误。而 JavaScript 数值只有一种数值数据类型，除了部分特殊情况，通常不会发生类型转换错误。从另一方面来说，由于数值类型能够与其他类型进行隐式转换，所以仍然存在大量陷阱[②]。

---

① 同理，对于 Java 和 JavaScript 中函数的参数和返回值，也存在类似的差别。JavaScript 的函数参数及返回值是不具有数据类型的。也就是说，静态数据类型和动态数据类型之间的差异在函数的类型中也会有所体现。

② 类型变换的问题将在之后详述。

布尔型在 Java 和 JavaScript 中是没有区别的，同样都是使用 true 和 false 的字面量。

null 类型的值只有 null 一种情况，并且 null 属于字面量。虽然 Java 中也存在 null 这一字面量，但是没有 null 类型，null 只是一种可以被引用的值而已。尽管有这样的差别，但是 Java 中的 null 和 JavaScript 中的 null 在用法上几乎没有区别，只需要注意一下类型转换的问题即可。

undefined 类型是指未定义的值的类型，这一概念在 Java 中是不存在的。如果在 Java 中使用类似于 undefined 类型的值的话，就会发生编译错误。

3.3 字符串型

3.3.1 字符串字面量

字符串值可以通过字符串字面量来表示。字符串字面量需要用双引号（"）或单引号（'）括起来。请看下面的例子。

```
js> var s = "abc";     // 将字符串值赋值给变量 s
js> print(s);          // 显示变量 s 的值
abc

js> var s = 'abc';     // 将字符串值赋值给变量 s
js> print(s);          // 显示变量 s 的值
abc
```

特殊字符可以通过转义字符（串）来表示。可以通过在转义符之后使用特定字符，来表达一些特殊的含义。转义符是反斜杠（\）。例如，\n 是换行符的表达方式。

表 3.1 总结了一些常用的转义字符（串）。

表 3.1　转义字符（串）

转义字符（串）	含义
\n	换行（LF）
\t	tab（水平制表符）
\b	退格
\r	换行（CR）
\f	换页
\v	垂直制表符
\\	反斜杠
\'	单引号
\"	双引号
\xXX	以十六进制代码 XX 表示的一个字符（X 是 0 到 9 的数字或 a 到 f 的字母）。例如，\x41 表示“A”
\uXXXX	以十六进制代码 XXXX 表示的一个 Unicode 字符（X 是 0 到 9 的数字或 a 到 f 的字母）。例如，\u03a3 表示希腊字符 Σ

因为可以用两种引号来包围字符串字面量，所以只要对此加以活用，就能够少用转义字符。例如，在包含大量双引号的字符串字面量外使用单引号，就能够减少转义字符的出现。

```
js> var s = 'I say "yes"'; // 对于双引号不需要使用转义字符（当然使用转义字符也没问题）
js> print(s);
I say "yes"
```

在一些脚本语言中，可以通过双引号和单引号来改变转义字符的作用。不过在 JavaScript 中，双引号和单引号除了有其各自的转义字符之外，在功能上并没有其他差异。

3.3.2 字符串型的运算

在前一节中可以看到，在 = 运算符右侧书写字符串值，就能够将其赋值给位于等号左侧的变量；如果在右侧书写的是值为字符串值的变量，也一样能将其值赋值给等号左侧的变量。仅靠语言描述可能不太容易理解，通过下面的例子就能一目了然了。

```
js> var s = 'abc';          // 将字符串值 'abc' 赋值给变量 s
js> var s2 = s;             // 将变量 s 的值赋值给 s2
js> print(s2);              // 变量 s2 的值为字符串值 'abc'
abc
```

熟悉“引用”或是“指针”的人看了上面的例子，也许会考虑下面的问题：当改变变量 s 的字符串值时，变量 s2 的值会随之改变吗？

这里先简单回答一下，之后再作详述。答案是，由于 JavaScript 的字符串型是不可变类型，所以字符串值本质上是不能改变的。这个答案看似没说什么，不过 JavaScript 确实和 Java 一样，其字符串型是不可变类型，所以这样的回答对于 Java 程序员来说，应该是不难理解的。

回到字符串运算符的话题。可以通过 + 运算符来连接字符串值。具体的例子如下所示。

```
js> var s1 = '012';
js> var s2 = '345';
js> var s3 = s1 + s2;       // 连接字符串值
js> print(s3);              // 变量 s3 的值为字符串值 '012345'
012345
```

+= 运算符可以在连接字符串的同时进行赋值。

```
js> var s = '012';
js> s += '345';
js> print(s);               // 变量 s 的值为字符串值 '012345'
012345
```

因为字符串值是不可变的，所以上面运算生成的是不同于 '012' 和 '345' 的新的字符串值 '012345'。在下面的例子中，变量 s2 的字符串值仍然是 '012'。

```
js> var s = '012';
js> var s2 = s;
js> s += '345';             // 变量 s 的值为字符串值 '012345'
js> print(s2);              // 变量 s2 的值仍保持为字符串值 '012'
012
```

可以通过 typeof 运算符来获知值的数据类型。对字符串型进行 typeof 运算的话，将会得到字符串值 "string" 这一结果。下面是一个具体的例子。

```
js> typeof 'abc';           // 对字符串字面量进行 typeof 运算
string

js> vas s = 'abc';
js> typeof s;               // 对变量 s 的值进行 typeof 运算
string
js> typeof(s);              // 也可以添加括号
string
js> typeof(typeof(s));      // typeof 运算的结果也是字符串值
string
```

3.3.3 字符串型的比较

JavaScript 有两种等值运算符，即 === 和 ==。与之对应，也有两种不等运算符 !== 和 !=。

=== 和 == 的区别在于，在比较的时候是否会进行数据类型的转换。=== 在比较的时候不会对数据类

型进行转换。在 ECMAScript 标准中，将其称为严格相等（strict equal）。详细的内容将在之后进一步说明，现在先讨论字符串比较的问题。如果只考虑字符串之间的比较，=== 和 == 的结果是没有区别的。两种方式都会判断字符串的内容是否一致。下面是具体的例子。

```
js> var s1 = '012';
js> var s2 = '0';
js> var s3 = s2 + '12';
js> s1 == s3;              // 字符串的内容是相同的
true
js> s1 === s3;             // 字符串的内容是相同的
true
js> s1 != s3;
false
js> s1 !== s3;
false
```

此外，还能够对 JavaScript 中的字符串值进行大小比较运算，为此分别有 > 运算符、>= 运算符、< 运算符和 <= 运算符。对字符串值的比较基于 Unicode 字符的编码值（编码位置）。请参见下面的例子。

```
js> var s1 = 'abc';
js> var s2 = 'def';
js > s1 < s2;
true
js> s1 <= s2;
true
js> s1 > s2;
false
js> s1 >= s2;
false
```

下面是比较大小时 Unicode 的编码位置的一些典型情况。要深入理解这部分的内容，需要有 Unicode 的相关知识，不过只要能记住以下的情况，就多少能进行运用了[①]。

- 英文字母是字典顺序（ABC 顺序）
- 英文的大写字母在小写字母之前
- 数字和符号在英文字母之前（不过有些符号是在英文字母之后的）
- 日文的平假名在片假名之前
- 日文的平假名和片假名都是字典顺序（あいうえお顺序）
- 日文浊音和半浊音的顺序则是按以下的规律排列：へ、ほ、ぼ、ぽ、ま
- 日文汉字在平假名和片假名之后
- 日文汉字的排列顺序视计算机的具体情况而定（有些是按照音读的字典顺序）

在实际中，只有英语单词的大小比较是有意义的。平假名字符串和片假名字符串的比较虽然也能勉强进行，不过在其中还包含有日文汉字的情况下，由于还要考虑计算机系统的差异，所以这样的比较对于用户来说并没有实际意义。

3.3.4　字符串类（String 类）

之前提到，在 JavaScript 中字符串型是一种内建类型。不过 JavaScript 的字符串也有容易使人混淆的地方，即除了内建类型的字符串之外还存在一个字符串类。

字符串类的名称为 String。JavaScript 中字符串型和 String 类的关系，大致相当于 Java 中数值型和包装类型（Number 类和 Integer 类）的关系。字符串型和 String 类之间也同样支持隐式类型转换。在 Java

① 中文汉字的情况则更为复杂，是按《康熙字典》的部首顺序以及笔画顺序进行排序的，有兴趣的读者可以进一步阅读相关材料。——译者注

中存在装箱和拆箱转换，在 JavaScript 的字符串型和 String 类之间也有着类似的转换。这一转换通常是隐式进行的。例如，可以像下面这样获取字符串值的字数。

```
js> var s = '012';
js> s.length;              // 在形式上类似于读取字符串值的属性
3
js> '012'.lenght;          // 在形式上类似于读取字符串字面量的属性
3
```

上述代码中，其内部发生了字符串值到 String 对象[①]的隐式数据类型转换。

表面上，这里的代码行为和 Java 相类似（在 Java 中也可以执行 "012".length()），但其实两者在语法意义上是不同的。Java 的字符串字面量生成的是 String 类的对象，所以这里的点运算符的含义其实是一般意义上的对类方法的调用。

另一方面，在 JavaScript 中书写 '012'.lenght 的话，（属于内建类型的）字符串值会先被隐式地转换为字符串对象，然后再读取字符串对象的 length 属性。

当然了，过分在意内部实现细节，反而违背了 JavaScript 编程的初衷。其实只要简单地按照表面上的样子，将（属于内建类型的）字符串值当作一种对象来使用也没有问题。

3.3.5　字符串对象

可以使用 new 运算符，来显式地生成一个字符串对象。在第 5 章中，将会进一步详细说明 new 运算符，这里先理解下面的例子就可以了。

```
var sobj = new String('abc');      // 生成字符串对象
```

不过，因为像之前代码中所述的那样，字符串值能够被隐式转换为字符串对象，所以在实际中几乎不使用 new 来生成字符串对象。

隐式类型转换也能反向进行。在下面的例子中，代码中 sobj 所引用的字符串对象被转换为了字符串值之后，通过 + 运算符对其进行了字符串的连接。

```
js> var s = sobj + 'def';  // 将字符串对象隐式转换为了字符串值
js> print(s);
abcdef
```

字符串值和字符串对象之间可以进行隐式类型转换。因此，一般来说并不需要在意值和对象之间的区别。不过正因看起来非常相似，所以会存在一些陷阱。例如，在判定两者是否相等上是有差异的。对象的相等运算，判断的是两者是否引用了同一个对象（而非两者的内容是否相同）。请看下面的例子。

```
js> var sobj1 = new String('abc');
js> var sobj2 = new String('abc');
js> sobj1 == sobj2;        // 虽然字符串的内容相同，但是并非引用了同一个对象，所以结果是 false
false
js> sobj1 === sobj2;       // 虽然字符串的内容相同，但是并非引用了同一个对象，所以结果是 false
false
```

下面的规则或许会让你感到有些不可思议：上面的两个字符串对象，在通过 + 与空字符串值连接之后，就会进行隐式数据类型转换而变为字符串值，从而结果也将发生变化。

```
// 继续之前的代码（以下只是用于说明的代码，实际中并不推荐这样使用）
js> sobj1 + '' == sobj2 + '';
true
js> sobj1 + '' === sobj2 + '';
true
```

对于字符串值和字符串对象的等值判断，如果使用的是会进行隐式数据类型转换的 == 运算，则只会

① 在本书中，String 类的实例被称为 String 对象或是字符串对象。

判定其内容是否相同，如果内容相同则结果为真。

```
js> var sobj = new String('abc');
js> var s = 'abc';
js> sobj == s;              // 进行数据类型转换的等值运算的结果为 true
true;
js> sobj === s;             // 不进行数据类型转换的等值运算的结果为 false
false
```

对于比较大小运算，字符串对象和字符串值一样，都是比较其字符串内容。因此可以认为，这时字符串值和字符串对象之间没有区别。

3.3.6　避免混用字符串值和字符串对象

在必要的时候，可以使用 typeof 运算来判别一个字符串是字符串值还是字符串对象。字符串对象的 typeof 运算结果为 "object"。

```
js> var sobj = new String('abc');
js> typeof sobj;
object
```

要防止混用字符串值和字符串对象是很简单的，只要不显式地使用 new String() 即可。也就是说，应该避免显式地生成字符串对象。

需要使用字符串值的时候，一般都使用字符串字面量。对于其余的情况，只要像下面那样，通过 String 函数进行显式的数据类型转换就足够了。

避免显式地生成字符串对象并不意味着要避免使用字符串对象，应该是积极地使用隐式数据类型变换，将字符串值转换为字符串对象。不需要特别考虑内部结构，仅仅从表面上来看，转换为字符串对象后，只要在字符串值之后写上点运算符和属性名，就能对字符串进行各种各样的操作了。

可以像下面这样对字符串值调用各种方法，这在实际使用中非常方便。后面 3.3.8 节介绍的一些其他方法也会以这样的形式被调用。

```
js> var s = 'abc';          // 返回字符串值下标为 1 的字符
js> s.charAt(1);
b

js> 'abc'.charAt(1);        // 对于字符串字面量也能像这样进行方法调用
b
```

3.3.7　调用 String 函数

通过 new 运算符调用字符串（这种做法称为构造函数调用）容易引起混淆。事实上，仅通过调用 String 函数就可以生成字符串值。一般来说，使用 String 函数是为了进行显式的数据类型转换。

```
js> var s = String('abc');
js> typeof s;               // 变量 s 的值是字符串型
string

js> var s = String(47);     // 由数值类型向字符串值类型的显式数据类型变换
js> print(s);
47
js> typeof s;               // 变量 s 的值是字符串型
string
```

3.3.8　String 类的功能

表 3.2 对 String 类的函数以及构造函数调用进行了总结。

表 3.2 String 类的函数以及构造函数调用

函数或是构造函数	说明
String([value])	将参数 value 转换为字符串值类型
new String([value])	生成 String 类的实例

表 3.3 对 String 类的属性进行了总结。可以通过类似于 String.fromCharCode(0x41) 这样的形式使用。

表 3.3 String 类的属性

属性名	说明
fromCharCode([char0[, char1, ...]]	将参数 value 转换为字符串值类型
length	值为 1
prototype	用于原型链

表 3.4 对 String.prototype 对象所具有的属性进行了总结。

表 3.4 String.prototype 对象所具有的属性

属性名	说明
charAt(pos)	返回下标 pos 位置字符的长度为 1 的字符串值。下标从 0 开始。如果超过了下标的范围，则返回空字符串值
charCodeAt(pos)	返回下标 pos 位置字符的字符编码。如果超过了下标的范围，则返回 NaN
concat([string0, string1, ...])	和参数字符串值相连接之后返回新的字符串值
constructor	引用一个 String 类对象
indexOf(searchString[, pos])	返回在字符串中第一个遇到的字符串值 searchString 的下标值。可以通过第二个参数指定搜索的起始位置。如果没有找到符合条件的结果，则返回 –1
localeCompare(that)	比较和本地运行环境相关的字符串。根据比较的结果分辨返回正数、0 或者负数
match(regexp)	返回匹配正则表达式 regexp 的结果
quote()	JavaScript 自定义的增强功能。在字符串外加上双引号之后返回这一新的字符串值
replace(searchValue, replaceValue)	将 searchValue（正则表达式或者字符串值）替换为 replaceValue（字符串或者函数）后返回经过替换后的字符串
search(regexp)	返回匹配正则表达式 regexp 的位置的下标
slice(start, end)	将参数 start 开始至 end 结束的字符串部分作为新的字符串值返回。如果 start 和 end 是负数，则返回从末尾逆向起数的下标值。
split(separator, limit)	根据字符串或是正则表达式形式的参数 separator 将字符串分割，返回相应的字符串值数组
substr(start[, length])	JavaScript 自定义的增强功能。返回从参数 start 开始长度为 length 的新字符串值。如果 start 是负数，则从末尾逆向起数
substring(start, end)	将参数 start 开始至 end 结束的字符串部分作为新的字符串值返回。其作用和 slice 相同，但是不支持以负数作为参数
toLocaleLowerCase()	将字符串中的所有字符转换为和本地环境相应的小写字符
toLocaleUpperCase()	将字符串中的所有字符转换为和本地环境相应的大写字符
toLowerCase()	将字符串中的所有字符转换为小写字符
toSource()	JavaScript 自定义的增强功能。返回用于生成 String 实例的字符串（即源代码）
toString()	将 String 实例转换为字符串值（并返回）
toUpperCase()	将字符串中的所有字符转换为大写字符
trim()	去除字符串前后的空白符
trimLeft()	JavaScript 自定义的增强功能。去除字符串左侧（头部）的空白符
trimRight()	JavaScript 自定义的增强功能。去除字符串右侧（尾部）的空白符
valueOf()	将 String 实例转换为字符串值并返回①

表 3.5 对 String 类的实例属性进行了总结。可以通过类似于 str.length 的形式来使用 String 对象。

表 3.5 String 类的实例属性

属性名	说明
（内部属性）	字符串值
length	字符串长度（字符数）

还可以像下面这样，通过数值属性获取指定下标的字符（不过这是 JavaScript 自定义的增强功能）。

① 返回该字符串对象的原始值。——译者注

其返回值是一个 String 对象。

```
js> var s = new String('abc');

js> print(s[1]);               // 下标为 1 的字符
b
js> print('abc'[2]);           // 由于有隐式数据类型转换，所以对字符串值也能进行这样的操作
c
```

3.3.9　非破坏性的方法

字符串对象和字符串值一样，是不可变的。也就是说，不能改写字符串的内容。正如表 3.4 所述，所有要改变字符串内容的方法，都会生成一个新的字符串对象然后将其返回。

```
js> var s = new String('abc');

js> var s2 = s.toUpperCase();  // 调用对象 s 的 toUpperCase 方法
js> print(s, s2);              // 对象 s 的内容不发生变化
abc ABC

js> s[0] = 'A';                // 即使是 JavaScript 独有的 [] 运算也不会改写字符串的内容
js> print(s);
abc
```

改变内部状态的方法被称为破坏性的方法。在之后将会详述的 Array 类中有大量破坏性的方法，与 String 类形成鲜明对比。一般来说，非破坏性的方法更好一些。不过在有些时候，非破坏性方法的效率会相对较低。更为详细的内容请参见 5.12 节。

3.4　数值型

3.4.1　数值字面量

在 JavaScript 中，数值的内部结构为 64 位的浮点小数。不过在实际的编程中，使用整数的情况会更多。不管内部构造如何，从 JavaScript 的代码上来看，只要是写为整数就能够作为整数使用，而不必考虑是否是浮点数的问题。因为所有的数值都是浮点小数，所以其运行效率多少会有些下降。因此对于比较注重运行效率的程序来说，JavaScript 可能并不是合适的选择。

由于整数型和浮点数型在使用上没有差别，所以不会发生和类型转换相关的错误。当然，浮点小数本身所具有的缺点依然存在。不过对于大部分现代程序设计语言来说，如果要使用小数，这些问题还是难以避免。

数值字面量的具体示例请参见表 3.6。

表 3.6　数值字面量的例子

具体示例	说明
0	整数
51	整数
-7	负整数（从语法上来看，负号是单目运算符）
0X1f	16 进制数（从 a 至 f 的字母部分也可以写作大写。0x 也可以用 0X 代替）
3.14	实数
.14	实数。与 0.14 相等
3e2	3 乘以 10 的 2 次方，即 300。e 也可以写作 E
3.14e2	3.14 乘以 10 的 2 次方，即 314
3.14e-2	3.14 乘以 10 的 -2 次方，即 0.0314

下面是使用数值字面量的代码示例。

```
js> var n1 = 1;          // 将数值 1 赋值给变量 n1
js> var n2 = 2;          // 将数值 2 赋值给变量 n2
js> n1 + n2;             // 将变量 n1 的值与变量 n2 的值相加
3
```

JavaScript 也支持 8 进制的数值字面量，但是 ECMAScript 标准并不支持。这仅仅是为了向下兼容性而保留的功能，不推荐在实际中使用。

可以通过 typeof 运算符来判断数值的类型。对于数值来说，typeof 运算符返回的结果是字符串值 "number"。

```
js> var n = 1;
js> typeof n;            // 对数值进行 typeof 运算
number

js> typeof 1;            // 对数值字面量进行 typeof 运算
number
```

3.4.2 数值型的运算

对于数值可以进行 +（加法）、-（减法）、*（乘法）、/（除法）四则运算。通过 % 符号则可以进行求模运算（即计算除法运算后的余数）。其他的运算类型请参见第 4 章。

需要注意的是，尽管从代码上来看进行的是整数运算，但其实在其内部进行的仍然是浮点数运算。例如，对 0 作除法并不会得到错误的结果，而是会得到一个特殊的数值。这点将在之后详述。

在 JavaScript 标准对象中有一个 Math 对象。该对象定义了圆周率 PI、自然对数的底数 E 等数学常量，以及一些相关的数学函数。例如，可以通过 Math.pow 函数计算 2 的 10 次方。更详细的内容请参见 5.21.2 节。

3.4.3 有关浮点数的常见注意事项

这里列举一些有关浮点数的常见注意事项。在使用其他程序设计语言中的 double 型或是 float 型时也同样应该注意这些问题。在其他的程序设计语言中由于需要开发者显式地使用浮点型数值，因此在使用时会更加注意。而在 JavaScript 中，由于总是会使用浮点小数，因此这些问题常常会被忽视。

首先，要说一些可能让不了解浮点数的人感到惊讶的事实。对于浮点数来说，有时候并不能正确地表达小数点以后的部分。实际上，能够正确表达一个数的值反而是一种例外，大部分情况下浮点数只能表达数值的近似值。

下面是一个非常著名的例子。即使是这样简单的运算，也无法获得正确的结果。

```
js> 0.1 + 0.2;                  // 0.1 与 0.2 的和并不是 0.3。
0.30000000000000004
js> (0.1 + 0.2) == 0.3          // 两者不一致。
false
js> (0.1 + 0.2) === 0.3         // 两者不一致。
false

js> 1/3                         // 1 除以 3 之后的近似结果。
0.3333333333333333
js> 10/3 - 3;                   // 这同样是近似值。
0.33333333333333
js> (10/3 - 3) == (1/3);        // 这两个近似值是不一致的。
false
js> (10/3 - 3) === (1/3);
false
```

对于浮点数来说，要执行正确的实数运算是不可能的。稍加注意就会发现，这并不是个应该如何避免的问题，而是从原理上就是无法回避的。对于整数的情况，则可以保证在 53 位的范围内能有正确的结果。因此，如果只是使用 53 位以内的整数的话，就不会有任何问题。

如果需要用到数值正确的实数，就必须使用类似于 Java 中的 BigDecimal 类的实数库。目前，JavaScript 并没有标准的实数库。不过也有特殊情况，如果是在 JVM 下使用 Rhino（一种基于 Java 的开源 JavaScript 实现）的话，就能够直接使用 Java 的 BigDecimal 类了。

3.4.4　数值类（Number 类）

正如存在字符串类（String 类），JavaScript 中也存在数值类（Number 类）。字符串值和字符串类在经过隐式数据类型转换之后，就基本能够以相同的方式使用，与此类似，经过数据类型转换之后，数值和数值对象也能被视为等价的。

例如，可以对一个数值调用下面这样的方法。和字符串对象的情况类似，在内部其实是隐式地生成了数值对象，不过也并不需要对此特别在意。

```
js> (1).toString();             // 为了区分小数点和点运算符而必须使用括号。
1

js> typeof (1).toString();      // 确认是否确实是从数值转换为了字符串。
string
```

可以通过 new 运算符来显式地生成数值对象。和字符串对象的情况类似，这将引起等值运算的混乱，所以如果没有特殊的理由，不建议使用显式的数值对象。

```
js> var nobj = new Number(1);
js> var nobj1 = new Number(1);
js> nobj == nobj1;              // 虽然值是相同的，但是所引用的对象不同，因而结果为 false
false
js> nobj === nobj1;             // 虽然值是相同的，但是所引用的对象不同，因而结果为 false
false

js> nobj == 1;                  // 会进行数据类型转换的等值运算结果为 true
true
js> nobj === 1;                 // 不会进行数据类型转换的等值运算结果为 false
false
```

由于会进行隐式数据类型转换，因此数值和数值对象在外表上是没有什么区别的。如有必要，可以通过 typeof 运算对其进行判断。对数值对象执行 typeof 运算的结果为 "object"。

```
js> var nobj = new Number(1);
js> typeof nobj;
object
```

3.4.5　调用 Number 函数

和 String 函数类似，以通常的方式调用 Number 函数的话，将返回相应的数值。在需要显式地进行数据类型转换的时候，可以使用 Number 函数。

```
js> var n1 = Number(1);
js> typeof n1;                  // 变量 n1 的值为数值
number
js> n1 == 1;
true
js> n1 === 1;
true

js> var n = Number('1');        // 从字符串值至数值型的显式数据类型转换
```

```
js> print(n);
1
```

如果参数无法被转换为数值类型，Number 函数返回的结果将是 NaN。使用 new 运算符（构造函数调用）的情况也是如此[①]。

```
js> var n = Number('x');
js> print(n);
NaN
js> typeof n;
number

js> var nobj = new Number('x');
js> print(nobj);
NaN
js> typeof nobj;
object
```

3.4.6　Number 类的功能

表 3.7 对 Number 类的函数以及构造函数调用进行了总结。

表 3.7　Number 类的函数以及构造函数调用

函数或是构造函数	说明
Number([value])	将参数 value 转换为数值类型
new Number([value])	生成 Number 类实例

表 3.8 对 Number 类的属性进行了总结。可以以类似于 Number.NaN 的方式来使用这些属性。

表 3.8　Number 类的属性

属性名	说明
prototype	用于原型链
length	值为 1
MAX_VALUE	64 位浮点小数所支持的正数最大值
MIN_VALUE	64 位浮点小数所支持的正数最小值
NaN	含义为 Not a Number 的值
NEGATIVE_INFINITY	表示负无穷大的值
POSITIVE_INFINITY	表示正无穷大的值

表 3.9 对 Number.prototype 对象的属性进行了总结。

表 3.9　Number.prototype 对象的属性

属性名	说明
constructor	引用一个 Number 类对象
toExponential(fractionDigits)	转换为指数形式的字符串值。fractionDigits 为小数点位数
toFixed(fractionDigits)	转换为小数点形式的字符串值。fractionDigits 为小数点位数
toLocaleString()	转换为和本地环境相对应的字符串值
toPrecision(precision)	转换为小数点形式的字符串值。precision 为有效数字
toSource()	JavaScript 自定义的增强功能。返回用于生成 String 实例的字符串（即源代码）
toString([radix])	将 Number 实例转换为字符串值。参数 radix 为其基数
valueOf()	将 Number 实例转换为数值

表 3.10 对 Number 类的实例属性进行了总结。

① 关于 NaN，将会在之后进行详细说明。

表 3.10　Number 类的实例属性

属性名	说明
（内部属性）	数值

3.4.7　边界值与特殊数值

可以通过 Number 对象的属性值来获知 64 位浮点数所支持的最大正值和最小正值。如果在其之前添加负号（运算符），就能够获得相应的最大负值和最小负值[①]。

```
js> Number.MAX_VALUE;
1.7976931348623157e+308

js> Number.MIN_VALUE;
5e-324

js> -Number.MAX_VALUE;
-1.7976931348623157e+308

js> Number.MIN_VALUE;
-5e-324
```

还可以通过 Number.MAX_VALUE.toString(16) 来获得相应的 16 进制数值，由于结果较长，这里就不列出了，有兴趣的读者可以自己再确认一下。

在 JavaScript 中，浮点数的内部结构遵循 IEEE754 标准。可以通过 Number 对象的属性值来获得在 IEEE754 中定义的一些特殊数值。表 3.11 对此做了总结。

表 3.11　浮点小数的特殊数值

标识符	含义
Number.POSITIVE_INFINITY	正无穷大
Number.NEGATIVE_INFINITY	负无穷大
Number.NaN	Not a Number

对其进行实际求值的结果如下。

```
js> Number.POSITIVE_INFINITY;
Infinity

js> Number.NEGATIVE_INFINITY;
-Infinity

js> Number.NaN;
NaN
```

从内部结构来看，这 3 个特殊数值（即正负无穷大与 NaN）都是基于 IEEE754 标准的比特位数值。虽然它们在形式上属于数值（typeof 运算符对它们的执行结果为 number），但是并不能作为数值进行计算。例如，将最大正数值乘以 2 之后能够得到正无穷大，但反之则不成立，正无穷大除以 2 之后无法得到最大正数值。

事实上，这 3 个特殊数值对于任何运算都无法得到通常的数值结果。

```
js> var inf = Number.MAX_VALUE * 2;
js> print(inf);                 // 将最大正数值乘以 2 之后能够得到正无穷大
Infinity

js> inf / 2;                    // 再除以 2 之后却无法得到原值
Infinity
```

① +0 和 −0 在语法规则上确实存在区别，不过在实际中并不需要对此加以区分（它们仅仅是为了定义正负无穷大而已）。

```
js> inf * 0;                    // 即使乘以 0，得到的结果也不是 0
NaN
```

NaN 是 3 个特殊数值中最为特别的，3.4.8 节中将对其进行单独说明。

3.4.8 NaN

对 NaN 进行任何运算，其结果都是 NaN。因此，如果在计算过程中出现了一次 NaN，最终的结果就一定会是 NaN。

```
js> NaN + 1;
NaN
js> NaN * 0;
NaN
js> NaN - NaN;
NaN
```

NaN 的运算还具有更为特别的性质。NaN 不但不与其他任何数值相等，就算是两个 NaN 的等值判断，其结果也为假。

```
js> NaN == 1;                   // 该表达式的结果显然为 false
false
js> NaN === 1;                  // 该表达式的结果显然为 false
false
js> NaN == NaN;                 // 该表达式的结果也是 false
false
js> NaN === NaN;                // 该表达式的结果依然是 false
false
```

NaN 对于各种比较运算的结果也总是

```
js> NaN > 1;
false
js> NaN >= 1;
false
js> NaN > NaN;
false
js> NaN >= NaN;
false
```

由此可见，对于值为 NaN 的数值是无法进行判断的。不过在 JavaScript 中预定义了一个全局函数 isNaN。isNaN 函数的返回值为真的条件是其参数的值为 NaN，或是在经过数据类型转换至数值类型后值为 NaN。

```
js> isNaN(NaN);
true

js> var nanobj = new Number(NaN);    // NaN 值的 Number 对象
js> typeof nanobj;
object
js> isNaN(nanobj);                   // 这里也为真
true

js> isNaN({});                       // 将其转换为数值类型之后值为 NaN
true
```

而预定义全局函数 isFinite 可以对 3 个特殊数值（即 NaN 与正负无穷大）之外的数值进行判断。

```
js> isFinite(1);
true;
js> isFinite(NaN);
false
js> isFinite(Infinity);
false
```

```
js> isFinite(-Infinity);
false
```

可以通过特定的运算来得到特殊数值结果（表 3.12）。不过在实际编程过程中，故意产生 Infinity 或是 NaN 的情况十分少见。事实上，不小心在运算过程中生成的特殊数值往往是发生错误的根源。

表 3.12　可以得到浮点数特殊数值的运算

x	y	x/y	x%y
通常数值	0.0	无穷大	NaN
通常数值	无穷大	0.0	x
0.0	0.0	NaN	NaN
无穷大	通常数值	无穷大	NaN
无穷大	无穷大	NaN	NaN

Infinity 和 NaN 都是预定义的全局变量，所以从语法规则的角度来看，其值是可以改变的（当然不推荐这么做）。不过即使改变了它们的值，只要进行表 3.12 中的运算，就能够很容易地恢复 Infinity 和 NaN 的初始值。

```
// 以下是 ECMAScript 第 5 版中的结果

js> NaN = 7;
js> print(NaN);
NaN

js> Infinity = 8;
js> print(Infinity);
Infinity
```

在 ECMAScript 第 5 版中，已将 NaN 和 Infinity 改为了只读变量，因此不能再对它们进行数值更改。需要注意的是，在对它们进行赋值的过程中并不会报错。

3.5 布尔型

3.5.1 布尔值

布尔型也被称为逻辑值类型或者真假值类型。布尔型只能够取真（true）和假（false）两种数值。除此以外，其他的值都不被支持。

下面是使用了布尔型的具体代码示例。

```
js> var flag = true;
js> print(flag);
true
js> print(!flag);                    // true 的否定为 false
false
js> print(!!flag);                   // 否定的否定为肯定
true
```

true 和 false 定义为了字面量。字面量的使用方法与在代码中使用数值的方法类似，将它们理解为等价的行为即可。也就是说，var flag = true 这一代码中 true 的地位，就和代码 var n =0 中数值 0 的地位相同。

对布尔值进行 typeof 运算的话，得到的结果为 "Boolean"。

```
js> typeof true;
Boolean
js> typeof false;
Boolean
```

关于布尔值运算的相关说明，请参见第 4 章。

3.5.2 布尔类（Boolean 类）

布尔类（Boolean 类）是一种布尔型的包装类型。其地位以及使用方法和 String 类以及 Number 类相同。

像下面这样对布尔值使用点运算符的话，就能够对布尔值进行隐式数据类型转换，将其转为布尔对象。但是，布尔类中并没有什么实用的方法，所以基本上也很少会去使用。

```
js> true.toString();              // 隐式数据类型转换为了 Boolean 对象
true
```

和 String 以及 Number 一样，它也可以通过 new 运算符显式地生成布尔对象。不过也与 String 和 Number 的情况类似，一般来说没有必要显式地生成布尔对象。

```
js> var t = new Boolean(true);    // 构造函数调用
js> t;
true
js> typeof t;                     // 布尔对象
object
js> t == true;                    // 进行数据类型转换的等值运算结果为 true
true
js> t === true;                   // 不进行数据类型转换的等值运算结果为 false
false
```

通过调用 Boolean 函数可以将任意值显式地转换为布尔值。不过根据具体的代码语境，必要时，一个值将会被隐式地转换为布尔值类型，所以一般来说，并不需要显式地进行数据类型转换。

```
js> var tval = Boolean(true);     // 通过函数调用来进行显式的数据类型转换
js> typeof tval;
Boolean
js> tval;
true
js> tval == true;
true
js> tval === true;
true
```

3.5.3 Boolean 类的功能

表 3.13 对 Boolean 类的函数以及构造函数调用进行了总结。

表 3.13 Boolean 类的函数以及构造函数调用

函数或是构造函数	说明
Boolean(value)	将参数 value 转换为布尔值类型
new Boolean(value)	生成 Boolean 类实例

表 3.14 对 Boolean 类的属性进行了总结。

表 3.14 Boolean 类的属性

属性名	说明
prototype	用于原型链
length	值为 1

表 3.15 对 Boolean.prototype 对象的属性进行了总结。

表 3.15 Boolean.prototype 对象的属性

属性名	说明
constructor	对 Boolean 类对象的引用
toSource()	JavaScript 自定义的增强功能。返回用于生成 Boolean 实例的字符串（即源代码）
toString()	将 Boolean 实例转换为字符串值
valueOf()	将 Boolean 实例转换为布尔值

表 3.16 对 Boolean 类的实例属性进行了总结。

表 3.16　Boolean 类的实例属性

属性名	说明
（内部属性）	布尔值
toSource()	JavaScript 自定义的增强功能。返回用于生成 Boolean 实例的字符串（即源代码）

3.6 null 型

null 值的意义存在于对象引用之中。null 值最初的含义为“没有引用任何对象”。null 型只能够取 null 这一个值。null 值是一个字面量。由于只支持 null 这个值，所以将 null 型称为一种类型未免有些奇怪。不过从语法规则上来看，null 型确实是一种数据类型。

然而，对 null 值进行 typeof 运算得到的结果也是 "object"（具体原因尚不得知）[①]。因此，尽管其他的基本数据类型都可以通过 typeof 运算来进行类型判断，但对于 null 型来说，就必须通过和 null 值的等值判断才能确定其类型。

```
js> typeof null;          // typeof 运算的结果为 'object'
object
```

null 型没有与之相对应的 Null 类。因此，如果像下面这样对 null 值进行点运算，就会产生 TypeError 异常。

```
js> null.toString();
TypeError: null has no properties
```

和其他程序设计语言一样，null 值可能引发各种各样的错误，其中大部分和数据类型转换以及一些运算有关。这部分内容将在之后的章节中再进行说明。

3.7 undefined 型

undefined 型只能够取 undefined 这一个值。对 undefined 值进行 typeof 运算，其结果为 "undefined"。

```
js> typeof undefined;     // typeof 运算的结果为字符串值 'undefined'
undefined
```

从代码上来看，undefined 值似乎和 null 值一样都是一种字面量。但实际上，它并非字面量，而是一个预定义的全局变量。

由于有这样的设定，也就不难理解，可以像下面这样对变量 undefined 进行赋值。当然，本身并不应该进行这样的操作。

```
// ECMAScript 第 5 版之前的结果

js> undefined = 'abc';    // 对名称为 undefined 的全局变量进行赋值
js> print(undefined);
abc
js> typeof undefined;
string
```

由于 undefined 在 ECMAScript 第 5 版中变为了只读变量，所以上面的赋值是无效的。但是需要注意的是，这样的赋值并不会引起错误。

```
// ECMAScript 第 5 版中的结果
```

① 也有人认为这是 JavaScript 本身的一个 bug。

```
js> undefined = 'abc';
js> print(undefined);
undefined
```

undefined 值

在本书中，将 undefined 型的值称为 undefined 值。这与将字符串类型的值称为字符串值是一样的道理。下面的说法听起来或许有些麻烦，但是应当认识到全局变量 undefined 与 undefined 值的关系其实就是这样的：首先有了 undefined 型的值，之后才将该值赋值给了全局变量 undefined。

将这个值本身称为 undefined 值，其实是有些不自然的，这就好比是在将数值 0 赋值给变量 i 之后，就将数值 0 称为 i 值一样。不过，这个有些麻烦的问题在实际中并没有什么影响。从结果上来看，与数值或是字符串值的情况不同，undefined 型的值就只有这一个了。因此，尽管会有些不自然，这里还是继续使用 undefined 这种说法。

null 是一种字面量而 undefined 是一个变量名，这并不是一种偶然。要使一个变量的值为 null，就必须将 null 以字面量的形式赋值给该变量。因此从语言规则的角度来说，null 必须是一种字面量。另一方面，undefined 值最多只能算是某个没有经过显式赋值的变量的初始值。所以根据字面含义，将其称为未定义值或是未初始化值都没有问题。

可以像下面这样对此进行确认。

```
js> var u;                    // 只是被声明了的变量
js> typeof u;                 // 该变量的值为 undefined 值
undefined
```

也就是说，从语法规则上来看，undefined 这一标识符并不是必需的。这是因为只需将全局变量 undefined 赋值给没有被赋值的变量就可以了。null 值指的是没有引用任何对象的状态，尽管从含义上来看是否定的，但仍然是有其含义的。而 undefined 值则不同，像它的字面意思那样，仅仅指的是一个尚未定义的值。

对于 undefined 值来说，并不存在与之相对应的 Undefined 类。因此如果像下面这样对 undefined 值进行点运算，将会产生 TypeError 异常。

```
js> undefined.toString();
TypeError: undefined has no properties
```

下面总结了会出现 undefined 值的情况。

- 未初始化的变量的值
- 不存在的属性的值
- 在没有传入实参而调用函数时，该函数内相应参数的值
- 没有 return 语句或是 return 语句中不含表达式的函数的返回值
- 对 void 运算符求值的结果（常常会通过使用 void 0 来获取一个 undefined 值）

上一节中的 null 值可能会引起程序出错，而 undefined 值比它更容易引发错误。令情况更为混乱的是，如果对 null 值和 undefined 值做进行数据类型转换的等值运算（==），结果为真。此外，对于不进行数据类型转换的等值运算（===），其结果则为假。

undefined 值是一种有着非常高的潜在出错风险的语言特性，所以在使用 undefined 值时请多加留心。

3.8 Object 类型

除了基本类型之外，其他的所有类型都是 Object 类型。之后将会详述对象的概念。这里只要先知道在 5 种基本数据类型之外还有一种 Object 类型即可。

对 Object 类型进行 typeof 运算，得到的结果是 "object"。

```
js> var obj = {};          // 生成空的对象
js> typeof obj;            // 对变量 obj 所引用的对象进行 typeof 运算
"object"
```

Function 类型

对于 JavaScript 中是否存在 Function 类型，人们的意见尚未统一。我的意见是，只要将 Function 类型认为是 Object 类型的一种类型就足够了。在此只需要了解，对一个函数进行 JavaScript 标准的类型判定方法，也就是其 typeof 运算的结果为字符串 "function"。

3.9 数据类型转换

JavaScript 这种语言很容易在进行数据类型转换时发生错误。因为不具有强数据类型，所以会有大量的隐式数据类型转换。JavaScript 会根据上下文语境，自动地进行数据类型转换。例如，无论对 if 条件语句使用怎样的值，该值都将被转换为布尔型。

语句中所写的值也会被转换为和运算符相对应的值。例如，某个值与字符串值和连接运算符（+ 号）相连的话，不管该值是哪种类型，它都将被自动转换为字符串型。

这样的隐式数据类型转换，虽然有不需要进行显式的数据类型转换，以及不会产生类型不一致错误的优点，但是也有不足，且其中很多错误只有在运行时才能被发现。

不过，要是由于过分担心隐式数据类型转换可能造成的问题，而始终使用显式的数据类型转换的话，又不符合 JavaScript 编程的风格。JavaScript 注重的是灵活运用隐式数据类型转换，以写出简洁的代码。虽然确实存在着不少陷阱，但在必要的时候，应灵活使用显式的数据类型转换。

3.9.1　从字符串值转换为数值

下面是一些从字符串值转换为数值的具体示例。

通常的做法是使用 Number 函数、parseInt 函数和 parseFloat 函数。Number 函数的书写最为简单，不过需要注意的是，对于像 '100x' 这样的包含非数字的字符串值，函数返回的结果将是 NaN。而 parseInt 和 parseFloat 将会忽略数字以外的其他字符，所以 '100x' 将被转换为 100。parseInt 函数还可以通过第二参数来指定转换时所采用的基数（radix）。如果省略该参数则默认进行 10 进制转换。

```
js> typeof Number('100');  // 将字符串值 '100' 转换为数值 100
number
js> Number('100x');
NaN
js> parseInt('100');
100
js> parseInt('100x');
100
js> parseInt('x');
NaN
js> parseInt('ff', 16);
255
js> parseInt('0xff', 16);
255
js> parseInt('ff', 10);
NaN
js> parseInt('0.1');
0
js> parseFloat('0.1');
```

```
0.1
```

接下来说明一下从字符串值到数值的隐式数据类型转换。只要在数值运算操作数的位置上书写字符串值，该值就将被隐式地转换为数值类型。下面是一些具体的例子。

```
js> '100' - 1;                // 字符串值 '100' 被转换为了数值 100
99
js> '100' - '1';              // 两个操作数位置的字符串值被转换为了数值
99
js> '100' - '';               // 之后将会说明将字符串转换为数值的惯用方式
100
```

如果是加法运算，则不一定能获得期望的结果。这是因为对于＋运算来说，如果操作数中含有字符串值，它就将变为字符串连接运算。于是发生的就不再是从字符串值到数值的数据类型转换，而是从数值到字符串值的数据类型转换了。这一点在下一节中也将进行说明。

```
js> '100' + 1;                // 不再是数值加法，而是字符串连接运算
1001
js> 1 + '100';                // 即使第一个操作数是数值，也是字符串连接运算
1100
```

＋运算在作为单目运算符的情况下则是正号运算。这时操作数将被转换为数值类型。不过由于正号运算没有任何实质意义，所以其作用就仅仅是将字符串值转换为数值。

```
js> typeof +'100';            // 将字符串值 '100' 转换为数值 100
number
js> var s = '100';
js> typeof +s;                // 将字符串值 '100' 转换为数值 100
number
```

3.9.2　从数值转换为字符串值

下面来看一下将数值转换为字符串值的情况。首先是显式数据类型转换的例子。通常的做法是使用 String 函数，或是在对数值对象进行了隐式数据类型转换之后，再对其调用 toString 方法。

```
js> typeof String(100);       // 将数值 100 转换为字符串值 '100'
string
js> typeof (100).toString();  // 将数值 100 转换为字符串值 '100'。为了区分小数点和点运算符而必须使用括号
string

js> var n = 100;
js> String(n);                // 将数值 n 转换为字符串值
100
js> n.toString();             // 将数值 n 转换为字符串值
100
```

接下来看一下隐式数据类型转换。在字符串运算的操作数位置上写数值的话，就将对其进行隐式数据类型转换。以下是具体的例子。＋运算符的操作数中如果含有字符串，则会作为字符串连接运算符使用。

```
js> 'foo' + 100;              // 数值 100 被转换为了字符串值 '100'
foo100
js> 100 + 'foo';              // 就算数值位于左操作数的位置也一样会被转换为字符串值
100foo

js> var n = 100;
js> 'foo' + n;                // 将数值 n 转换为字符串值
foo100
```

3.9.3　数据类型转换的惯用方法

正如之前所介绍的，可以通过多种方式来实现数据类型转换。具体哪种方法的运行速度更快和具体

的实现有关，不能一概而论。此外，对于客户端 JavaScript 的情况，重要的不仅仅是需要考虑代码的运行速度，还应当尽可能地缩短源代码的长度。只有缩短代码长度才能够减少网络传输的时间。因此，以较短的代码长度实现数据类型转换的写法成为了首选。

下面是最为简短的数据类型转换写法。虽然使用 String(n) 或 Number(s)，代码可读性可能会更高一些，不过这样的写法能够实现更好的性能。

```
// 惯用的数据类型转换方法（最简短的写法）

// 从数值转换为字符串值
js> var n = 3;
js> n+'';                    // 将数值 3 转换为字符串值 '3'（利用了字符串连接运算符）

// 从字符串值转换为数值
js> var s = '3';
js> +s;                      // 将字符串值 '3' 转换为了数值 3（利用了正号运算）
```

表 3.17 列出了在字符串值和数值之间进行类型转换时需要注意的地方。隐式数据类型转换虽然简单，但容易引发错误。最为典型的错误就是在将字符串值转换为数值类型时，其结果可能会是 NaN。

正如 3.4.8 节所述，一旦在运算中出现了 NaN，整个数值运算的结果都将变为 NaN。这样不仅无法得到正确的结算结果，而且无法通过逆向的数据类型转换来得到原来的字符串。

表 3.17　在字符串值和数值之间进行数据类型转换时需要注意的地方

转换的对象	数据类型转换	结果
无法被转换为数值的字符串值	转换为数值类型	数值 NaN
空字符串值	转换为数值类型	数值 0
数值 NaN	转换为字符串型	字符串 "NaN"
数值 Infinity	转换为字符串型	字符串 "Infinity"
数值 –Infinity	转换为字符串型	字符串 "–Infinity"

3.9.4　转换为布尔型

在实际编程中，从其他类型向布尔型的数据类型转换是很重要的。在 if 语句或是 while 语句等的条件表达式中，会有很多这样的隐式数据类型转换。下面列举了在类型转换后结果为 false 的值。除此之外的值都将被转换为 true。

- 数值 0
- 数值 NaN
- null 值
- undefined 值
- 字符串值 ''（空字符串值）

请看下面的例子，下面的代码中在 if 语句的条件表达式里写了数值 0。在经过隐式数据类型转换之后，它将变为布尔型的 false。

```
js> if (0) { print('T'); } else { print('F'); }
F
```

尽管也可以通过 Boolean 函数，将一个值显式地转换为布尔型。不过通常来说，都是像下面这样，使用 !! 来进行隐式的数据类型转换的。

! 运算是用于布尔型操作数的逻辑非运算。在操作数不是布尔型的情况下会自动将其转换为布尔型。因此只要使用 !! 这样的双重否定，就能够将值转换为布尔型。

```
js> !!1;
true
```

```
js> !!'x';
true
js> !!0;
false
js> !!'';
false
js> !!null;
false
```

在进行布尔型的数据类型转换时，应当对 Object 类型的情况多加注意。Object 类型在被转换为布尔型之后结果必定为 true。以下代码的结果都是显示 T。这和直观的感觉是有所不同的，所以请务必注意。

```
js> var b = new Boolean(false);
js> if (b) { print('T'); } else { print('F'); }
T

js> var z = new Number(0);
js> if (z) { print('T'); } else { print('F'); }
T

js> var s = new String('');
js> if (s) { print('T'); } else { print('F'); }
T
```

而像下面这样，通过函数调用方式获得的结果就不再是 Object 类型，而是相应的内建类型了。这样一来，转换的结果就会与直觉一致。

```
js> var b = Boolean(false);
js> if (b) { print('T'); } else { print('F'); }
F

js> var z = Number(0);
js> if (z) { print('T'); } else { print('F'); }
F

js> var s = String('');
js> if (s) { print('T'); } else { print('F'); }
F
```

3.9.5　其他的数据类型转换

表 3.18 总结了对布尔值、null 值和 undefined 值进行数据类型转换时的情况。

表 3.18　对布尔值、null 值和 undefined 值进行数据类型转换的情况

被转换的值	转换为数值型	转换为字符串型
true	1	'true'
false	0	'false'
null 值	0	'null'
undefined 值	NaN	'undefined'

3.9.6　从 Object 类型转换为基本数据类型

下面总结了从 Object 类型转换为基本数据类型的规则，以及进行显式数据类型转换的方法（表 3.19）。其中变量 obj 是某一对象的引用。

表 3.19　对 Object 类型进行数据类型转换

转换后的类型	显式数据类型转换方法	说明
字符串型	String(obj)	将 toString() 方法的结果转换为字符串型

（续）

转换后的类型	显式数据类型转换方法	说明
数值型	Number(obj)	即 valueOf 方法的结果。如果 valueOf 方法的结果无法被转换为数值型，则改将 toString 方法的结果转换为数值型
布尔型	Boolean(obj)	总是 true
undefined 值	NaN	'undefined'

下面是将 Object 类型分别显式地与隐式地转换为字符串型的示例。

```
js> var obj = {};           // 生成空对象
js> obj + '';               // 将对象隐式地转换为字符串型（通过字符串连接运算符）
[object Object]
js> String(obj);            // 将对象显式地转换为字符串型
[object Object]
```

上面的结果是调用了对象 obj 的 toString 方法而得到的。如果像下面这样，改变了 toString 方法的实现的话，结果也将随之发生变化。

```
js> obj.toString();         // 确认当前 toString 方法的结果
[object Object]

js> obj.toString = function() { return 'MyObj'; }    // 重写 toString 方法的实现
js> obj.toString();         // 确认 toString 方法的结果
MyObj

js> obj + '';               // 将对象隐式地转换为字符串型
MyObj
js> String(obj);            // 将对象显式地转换为字符串型
MyObj
```

3.9.7　从基本数据类型转换为 Object 类型

下面总结了从基本数据类型转换为 Object 类型的规则（表 3.20）。

表 3.20　转换为 Object 类型

被转换值的类型	数据类型转换的结果
字符串型	String 对象
数值型	Number 对象
布尔型	Boolean 对象
null 型	Error 对象
undefined 型	Error 对象

和基本数据类型之间的类型转换一样，Object 类型和基本数据类型之间的数据类型转换，也可以根据上下文语境隐式地进行。因此，实际编程中，并不太需要进行显式的数据类型转换，反倒是更应该注意避免由于隐式数据类型转换而引发的错误。

随意举一例来说，下面的代码并不会报错，而是会将变量 obj 的值转换为 NaN。

```
js> var obj = {};
js> obj++;                  // 该对象被隐式地转换为了数值型
js> obj;                    // 变量 obj 的值为 NaN
NaN
```

上面的代码中，++ 运算将操作数转换为了数值型。

如果一个对象没有能返回恰当数值的 valueOf 方法，或是能返回可以被转换为恰当数值的字符串的 toString 方法的话，它在被转换为数值型之后的结果将是 NaN。又因为对 NaN 进行 + + 运算的结果为 NaN，所以变量 obj 的最终值将是 NaN。

专栏

JavaScript 的性能分析工具

使用性能分析工具，就能了解一个运行中的 JavaScript 程序正在执行哪一个函数、正在执行哪一行，以及花费了多少时间。使用性能分析工具的目的通常是为了检查程序中运行速度较慢的部分。这样的部分被称为运行瓶颈。根据经验，如果程序的运行很慢，很有可能是由于其中某一部分花费了较多的运行时间。因此，如果提高了瓶颈部分的执行速度，程序整体的运行性能也很可能得以提高。反过来说，如果不确定程序的瓶颈所在，就胡乱地进行性能优化，效果往往差强人意。

程序的执行时间并不仅仅由一些代码的处理时间决定。对于客户端 JavaScript 来说，以下几个因素相结合最终决定了用户的实际使用感受。

- **JavaScript 代码的执行时间（这也是仅凭性能分析工具所能测得的）**
- **DOM 的渲染时间**
- **网络响应时间**

Firebug、IE 的开发者工具和 Chrome 的开发者工具都具有基本的性能分析功能。此外，表 A 还列举了一些包含了其他各方面的分析测试功能的性能分析工具。

表 A　性能分析工具

名称	支持的浏览器	URL
YSlow	Chrome/Firefox	http://developer.yahoo.com/yslow/
Page Speed	Chrome/Firefox	http://code.google.com/speed/page-speed/
dynaTrace	Firefox/IE	http://ajax.dynatrace.com

第 4 章 语句、表达式和运算符

本章将总结 JavaScript 的语法规则。尽管 JavaScript 在语法结构上，有不少地方和 Java 类似，但它有一些自己独有的语句。同样地，在 JavaScript 中，有很多和 Java 相似的运算符和表达式。不过，因为隐式的数据类型转换在 JavaScript 中非常普遍，所以与 Java 相比，在使用这一语言的过程中还有其他一些必须注意的地方。

4.1 表达式和语句的构成

JavaScript 的源代码本质上是一个语句的集合。语句是由语句和表达式所构成的。表达式则由表达式和运算符所构成。这种在自身的定义中递归地使用自身的定义方式，在程序设计语言中相当常见。

有人可能会觉得，这种使用了自身的定义方式即使在经过了无限次循环之后，也无法真正地定义出一个概念。不过事实上，语句和表达式都具有不需要用到自身定义的定义方式。因此，这种递归的定义是不会无限循环下去的。对于语句来说，最终都可以被分解为保留字（之后将详述）、表达式与符号（括号或是分号等）。也就是说，即使在一条语句中包含其他语句，只要对这条被包含的语句继续进行分解，最终都会到达仅包含保留字、表达式与符号的状态。对于表达式来说，虽然也能在一句表达式中包含其他的表达式，不过只要对所包含的表达式继续进行分解，最终总是能达到仅包含标识符（变量名或是函数名）、字面量（即直接写出其值的数值或是字符串）与运算符（符号或是保留字）的状态。

4.2 保留字

在 ECMAScript 第 5 版中，保留字（reserved word）是按照如下方式分类的（表 4.1）。

表 4.1　JavaScript 的保留字

名称	说明
关键字	请参见表 4.2
今后的保留字	请参见表 4.3
null	字面量
true	字面量
false	字面量

表 4.2 与表 4.3 列出了在 ECMAScript 中所定义的关键字或是为以后预留的保留字。

表 4.2　关键字

break	do	instanceof	typeof
case	else	new	var
catch	finally	return	void
continue	for	switch	while
default	if	throw	delete
in	try		

表 4.3 今后的保留字

class	enum	extends	super	const	export
import	implements	let	private	public	yield
interface	package	protected	statics		

4.3 标识符

标识符是开发者在程序中所定义的单词，例如变量名或是函数名。虽说标识符中可以使用的字符是有所限制的，不过只要不与保留字中的单词重复就没有问题，所以实际上可以生成无限多的标识符。其具体的命名规则如下。

- 必须是除保留字以外的单词。
- 必须是除 true、false、null 以外的单词。
- 必须是以 Unicode 的（非空）字符开始，之后接有 Unicode 字符或是数字的单词。
- 单词的长度并无限制。

不能使用和保留字相同的单词作为标识符。例如，如果有一个函数被命名为 do，那么就会引起 SyntaxError。不过，如果仅仅是在标识符的字符中包含了保留字就没有问题，例如 doit 这样的函数名就是合法的。

true、false 和 null 这三个单词是字面量，不能被用作标识符。就好比数值字面量 1 或是字符串字面量 "abc" 不能被用作标识符一样，这三个字面量也不能被用作标识符。

JavaScript 中的标识符都是以 Unicode 字符所组成的单词。Unicode 字符中包含了日文字符（日文汉字以及平片假名），所以从语法上来说，是可以使用日文作为变量名以及函数名的。不过，由于一些历史原因以及习惯用法，并不推荐使用日文作为标识符[①]。在实际的编程过程中应当遵循以下的规则，即应该使用以英文字符（大写或是小写的英文字符）、_（下划线字符）或是 $（美元字符）开始，之后接有英文字符、_、$、数字（0 至 9）的单词。

下面是一些标识符的具体例子。由于 JavaScript 区分英文字符的大小写，所以 foo 和 Foo 将会被识别为不同的标识符。

- foo
- Foo
- FOO
- foo1
- foo_1
- _foo
- $
- $foo

习惯上，以下划线（_）开始的标识符会被作为“内部标识符”来使用。又因为在 prototype.js 中，getElementById 函数的别名被记为了 $，所以一些常用名称的别名常常会使用以美元符号（$）开始的标识符。

① Unicode 字符中也包含了繁简体中文汉字等各类中文字符，同样不推荐在标识符中使用中文字符。——译者注

4.4 字面量

字面量（literal）指的是，在代码中写下这些值之后，将会在运行时直接使用这些值的字面含义。有读者也许会觉得，在代码中书写的值自然会在运行时按原样表达该值，不过事实上并非如此，请看下面的代码。

```
// 字符串字面量 "bar" 的例子
var foo = "bar";
```

根据语法规则，代码中的 var 这个词的含义是变量的声明，因此，在运行中 var 并不会被识别为一个内容为 var 的单词。类似地，foo 这个词在运行时也不会被识别为一个内容为 foo 的单词，而仅被认为是变量 foo 所表示的值。而即使把代码中所有的 foo 都改写为 foo2 也不会改变运行结果，通过这一事实也能进一步理解该规则。

另一方面，"bar" 是一个字符串字面量，所以 bar 这一单词在运行过程中的含义就是 bar 这一字符序列而已。

数值字面量的情况就更加容易理解了。在下面的代码中写有两个数值 0。val0 中的 0 是其变量名的一部分，并不具有数值 0 的含义。这个 0 已经失去了可以进行算术运算的性质，仅仅是一个符号。

另一方面，右侧的字面量 0 则具有数值的含义。

```
// 数值字面量 0 的例子
var val0 = 0;
```

表 4.4 对未在本书中说明的 JavaScript 字面量进行了总结。

表 4.4　字面量

名称	具体示例
数值	100
字符串值	"foobar"
布尔值	true
null 值	null
Object	{ x:1, y:2 }
数列	[3, 1, 2]
函数	function() { return 0; }
正则表达式	/foo/

4.5 语句

在程序设计语言中，语句（statement）的定义可以由该语言经过明确定义的语法（syntax）规则得到，并且可以在运行程序时执行（execute）语句。换一种角度来说的话，所谓运行一个程序，指的就是执行程序中一条条的语句[①]。

虽然说，源代码中的语句并不一定是和运行中的每一步一一对应的，不过考虑到程序在运行时确实是在逐一执行语句，所以在概念上并不矛盾。

在 JavaScript 中，语句之间使用分号分隔。严格来说，分号只是一部分语句的结尾。例如对于表达式语句，其结尾必须使用分号；而对于复合语句，其结尾则是不需要分号。对于这方面的规则，JavaScript 和 Java 基本相同，不过 JavaScript 对于分号的使用限制更为宽松。例如在 JavaScript 中，换行时所省略的

① 这里的说明更接近于过程式程序设计语言中的说法，对此请加以注意。

分号将被自动补全。本书的代码范例中并不会省略分号。如需了解在分号自动补全过程中可能会引发的问题，请参考其他书籍。

4.6 代码块（复合语句）

代码块是在大括号（{}）中所写的语句，以此将多条语句的集合视为一条语句来使用。这样一来，从语法上来说，代码中所有能够书写语句的地方都可以书写多条语句。代码块的例子在本书中有很多，这里就不再举例了。

值得注意的是，JavaScript（准确地说是 ECMAScript）的代码块中的变量并不存在块级作用域这样的概念。详细情况请参见 6.4 节。

4.7 变量声明语句

变量声明语句的格式为，在关键字 var 之后跟上所需的变量名。在多个变量名之间使用逗号（,）分隔的话，就能够同时声明多个变量。而使用 = 运算符，就可以在声明的同时对变量进行初始化。有关变量的详细说明，请参见第 5 章。

```
// 变量声明的例子
var foo;
var foo, bar;                       // 同时声明多个变量
var foo = 'FOO', bar = 'BAR';       // 在声明变量的同时进行初始化
```

4.8 函数声明语句

JavaScript 中的函数声明语句，和 Java 中方法的定义语句在语法上是基本相同的，不同之处在于，函数声明语句并不是以返回值类型开始，而是使用了关键字 function，并且在 JavaScript 中不用参数指定类型。

尽管在 ECMAScript 标准中，函数声明语句并没有被视为语句的一种，不过在本书中暂不考虑这种严格的语言规则，而将函数声明也视为一种语句。

```
// 函数声明语句的语法

function 函数名（参数，参数，……）{
    语句
    语句
    ……
}
```

函数名和参数的位置上所书写的是标识符。参数的数量没有限制，所以即使一个参数也没有关系。大括号中的是函数体，里面可以书写多条语句。

4.9 表达式语句

所谓表达式语句就是一条内容为表达式的语句。之后还会再次对表达式进行说明。JavaScript 不同于 Java 那样，Java 只有一部分的表达式能够被作为语句使用，而在 JavaScript 中，所有的表达式都可以被

视为一条表达式语句。不过很可惜，JavaScript 的这一特性并不是一个优点。

例如，下面这样的无意义的代码并不会引发错误。这是因为相等运算符（==）的表达式从语法上来说属于是表达式语句。

```
// 虽然没有意义，但是语法上并没有错误的代码

var a;
a == 0;        // 一条表达式语句（但是没有任何的实际效果）
```

执行上面的表达式语句并不会有任何效果。又因为不会引起语法错误，所以即使不小心把 == 错写成了 = 也不容易被发现。而在 Java 中，这类没有意义的表达式语句将会引起编译错误。能够很容易地发现问题，反而是一种优点。

前面已经举了一个没有意义的表达式语句的例子，而在有意义的表达式语句中，具有代表性的就是赋值表达式和函数调用表达式了。下面是实际的例子。

```
// 表达式语句的例子

var s;
s = 'foo';            // 赋值表达式的表达式语句
print(s);             // 函数调用表达式的表达式语句
```

4.10 空语句

仅含有分号的语句就是空语句。仅在一部分场合下空语句才有其使用价值。请看下面的代码。

```
// 空的代码块
while （条件表达式） {
}

// 包含了空语句的代码块
while （条件表达式） {
    ;
}

// 仅有空语句
while （条件表达式）
    ;
```

在上面的代码中即使不写空语句，在意义和语法上也都不存在问题。尽管如此，还依然写了空语句的理由，是为了告诉代码的阅读者，自己并非忘了在代码块中书写语句，而是特意写了这段不会进行任何操作的代码。

4.11 控制语句

有一类语法规则被称为控制语句。控制语句包括条件分支、循环、跳转（包括异常处理）这 3 类。如果没有这样的控制语句，JavaScript 在理论上，是按照源代码上所写的代码顺序从上至下地执行。这种执行方式被称为“顺序执行”。

有了控制语句之后，就可以实现顺序执行以外的代码执行方式。

4.12 if-else 语句

if 语句和 if-else 语句的语法结构如下。其中的条件表达式和语句不能省略。

```
// if 语句的语法

if （条件表达式）
    语句
```

```
// if-else 语句的语法

if （条件表达式）
    语句
else
    语句
```

与 if 对应的条件表达式及语句统称为 if 子句，而与 else 对应的条件表达式与语句则统称为 else 子句。可以把 if 表达式看作 if-else 表达式省略了 else 子句的特殊情况。因此在下文中将两者统一作 if-else 语句处理。

在条件表达式的位置所写的式子，将被求值并转换为布尔型。这一隐式的数据类型转换常常会带来各种各样的错误。对于布尔型的数据类型转换中需要注意的地方，请参见 3.9.4 节进行确认。

下面是一个 if-else 语句的具体示例。

```
// if-else 语句的例子
if (i == 0)
    print("if clause");
else
    print("else clause");
```

如果变量 i 的值为 0，则输出 "if clause"，否则，输出 "else clause"。

在 if 子句以及 else 子句中可以书写任意的语句。又因为 if-else 语句本身也是语句的一种，所以也能够在 if-else 语句的子句中再次使用 if-else 语句。下面是一个在 if 子句中使用 if-else 语句的例子。

```
// 嵌套 if-else 语句
if (I == 0)
    if (j == 0)
        print("i==0 and j==0");
    else
        print("i==0 and j!=0");
else
    print("i!=0");
```

上面这段代码的执行结果和所预期的结果是一致的。接下来再来考虑一下没有外层 else 子句的嵌套 if-else 语句的情况。如果保持原有的缩进情况不变，将得到如下的代码。

```
// 嵌套 if-else 语句
if (i == 0)
    if (j == 0)
        print("i==0 and j==0");
    else
        print("i==0 and j!=0");
```

也可以对其缩进进行修改，得到如下代码。

```
// 容易混淆的缩进
if (I == 0)
```

```
    if (j == 0)
        print("i==0 and j==0");
else
    print("i==0 and j!=0");
```

在上面的代码中，从缩进方式来看，似乎 else 子句是与外层的 if 子句（即条件表达式为 i==0 的 if 子句）相对应的。但事实上，缩进对代码的实际意义不会产生影响。换句话说，在上述两段代码中，必然存在一组代码，其实际执行方式与代码缩进格式所暗示的方式有所不同。对于这个问题的回答是，由于 JavaScript 中有 else 子句必定与最邻近的 if 子句相结合的规则，因此在本例中，else 子句是和内层的 if 子句（即条件表达式为 j==0 的 if 子句）相对应的。这也就意味着，第二组代码的缩进方式与其实际的执行方式是不相符的。

为了避免产生这样容易混淆的情况，可以使用支持自动缩进的文本编辑器。不过，其实只要始终使用代码块来书写 if 子句和 else 子句的话，就能够避免这一问题了，所以，在此更加推荐使用这种通用性更高的解决方式。

```
// 通过代码块来避免出现容 5 易使人误解的缩进
if (i == 0) {
    if (j == 0) {
        print("i==0 and j==0");
    } else {
        print("i==0 and j!=0");
    }
}
```

在本书中，即使 if 子句及 else 子句的内容只占 1 行，也始终会使用代码块的形式。根据这一原则，如果要让 else 子句对应外侧的 if 子句，就需要像下面这样书写。习惯了这种方式的话，通过大括号就能够很清楚地理解语句的结构。

```
// 通过代码块来避免出现容易使人误解的缩进
if (i == 0) {
    if (j == 0) {
        print("i==0 and j==0");
    }
} else {
    print("i==0 and j!=0");
}
```

不过这一方式也存在一种例外。试考虑在 else 子句中书写 if-else 语句的情况。如果还是按照始终使用代码块的原则来书写代码的话，就会变成下面这样的情况。

```
// 在使用了代码块之后代码变得有些冗长的例子
if (i == 0) {
    print("i==0");
} else {
    if (i == 1) {
        print("i==1");
    }
}
```

这样的写法虽然也没有什么错误，不过根据书写习惯，在 else 子句中书写 if-else 语句时最好按照以下形式。

```
// 根据习惯，上一段代码通常应该像下面这样来书写
if (i == 0) {
    print("i==0");
} else if (i == 1) {
    print("i==1");
}
```

同时，对于下面这样在 else 子句中有连续多个 if-else 语句的情况，这一习惯写法也是很方便的。习

惯之后会发现，条件表达式和执行语句的对应关系非常清晰，语句的可读性大为提升。

在下面的代码中，将会对变量 i 的值为 0、为 1 以及为 2 的几种分支情况依次进行判断。

```
// 习惯之后非常易于阅读的代码写法（不过，这时也应该考虑是否需要采用接下来将要说明的 switch-case 语句）
if (i == 0) {
    print("i==0");
} else if (i == 1) {
    print("i==1");
} else if (i == 2) {
    print("i==2");
} else {
    print("other");
}
```

4.13 switch-case 语句

switch-case 语句是一种语法结构与 if-else 有所不同的条件分支判断语句。其语法结构如下。

```
// switch-case 语句的语法

switch （语句） {
case 表达式 1:
    语句
    语句
    ……
case 表达式 2:
    语句
    语句
    ……
case 表达式 N:
    语句
    语句
    ……
default:
    语句
    语句
    ……
}
```

根据习惯，“case 表达式：”部分被称为 case 标签，而“default:”部分被称为 default 标签。从语法规则的角度来看，它们与 if-else 语句中子句的功能不同，起到的是跳转目标的作用。在 switch 语句中可以书写多个 case 标签，而 default 标签则只能使用 1 次。此外，default 标签是可以省略的。

尽管 JavaScript 中的 switch 语句在语法结构上与 Java 的相同，但它们在实际的语法规则上却有着一些细微的差异。在 JavaScript 中，switch 的括号内可以写任意类型的表达式，case 标签中也可以写任意的表达式。与之相对应地，在 Java 的 case 标签中，则只能使用在编译时就能够获得结果的常量表达式。

switch 语句会把其在 switch 之后的括号内的表达式，与 case 标签中所写的各个表达式，依次通过相等运算符（===）进行比较。为了避免用词混淆，这里将前者称为 switch 表达式，而将后者称为 case 表达式。=== 运算符是不会进行数据类型转换的相等运算符。switch 语句首先对 switch 表达式进行求值，之后依次对 case 表达式从上往下求值，并将其结果与 switch 表达式的求值结果进行等值比较（===）。如果值相等，则跳转至该 case 标签处。如果与所有的 case 表达式都不等值，则跳转至 default 标签处。

下面的例子主要展示了一些在 Java 中不被允许的类型（代码清单 4.1）。

代码清单 4.1　switch 语句的例子

```
var s = 'foo';

switch (s) { // 可以在 switch 表达式中使用字符串值。
// 可以在 case 表达式中使用和 switch 表达式类型不同的值。
// s === 0 的值为假，所以将继续进行比较。
case 0:
    print('not here');
    break;

// 可以在 case 表达式中使用含有变量的表达式。
// s === s.length 的值为假，所以将继续进行比较。
case s.length:
    print('not here');
    break;

// 可以在 case 表达式中使用方法调用表达式。
// s === (0).toString() 的值为假，所以将继续进行比较。
case (0).toString();
    print('not here');
    break;

// 还可以在 case 表达式中书写这样的表达式。
// s === 'f' + 'o' + 'o' 为真，所以将执行以下的代码。
case 'f' + 'o' + 'o':
    print('here');
    break;

// 如果所有的 case 表达式在等值运算 ( === ) 后得到的结果都为假，则执行以下的代码。
default:
    print('not here');
    break;
}
```

乍一看，case 标签之间的部分是作为一个整体来执行的，不过实际上，case 标签并没有对代码按块进行分割的功能。因此在一个 case 标签结束执行之后，并不会跳出 switch 语句。

在代码清单 4.2 的 switch 语句中，虽然第一个 case 标签的比较结果就为真，但之后所有的 case 标签也都会被执行。

代码清单 4.2　没有 break 语句的 switch 语句，将不会在执行完其中某一段 case 之后就结束整个 switch 语句

```
var x = 0;
switch (x) {
case 0:
    print("0");
case 1:
    print("1");
case 2:
    print("2");
default:
    print("default");
    break;
}
```

```
// 代码清单 4.2 中 switch 语句的执行结果
0
1
2
default
```

由这一结果可知，应该将 case 标签与 default 标签看作跳转的目标地址。也就是应该这样来理解 case 标签的作用：当 switch 表达式与 case 表达式的值相一致时，就跳至该 case 标签所在位置，执行完该段代码之后继续从上往下逐次执行后续语句。

在很多时候，上面这样的代码都无法得到预期的结果。像代码清单 4.3 这样使用 break 语句就可以强制跳出当前 switch 语句。

代码清单 4.3 通过 break 语句跳出 switch 语句

```
var x = 0;
switch (x) {
case 0:
    print("0");
    break;
case 1:
    print("1");
    break;
case 2:
    print("2");
    break;
default:
    print("default");
    break;
}
```

```
// 代码清单 4.3 的运行结果
0
```

可以说，代码清单 4.3 是一种良好的 switch 语句使用模板。事实上，不使用 break 语句的 switch 语句才是比较少见的。

如果在 if-else 语句中有多个连续的分支判断，则可以将代码改写为等价的 switch 语句。在两种写法都可以的情况下，具体选择哪一种只是偏好问题，认为哪一种方式的可读性更好就选择哪一种即可。根据情况不同，有时比起 if-else 语句，switch 语句的可读性会更好（这样说可能带有一些主观偏好）。至少在使用 switch 语句时，等值比较表达式可以被隐藏起来，所以与使用了等值比较的 if-else 语句相比表达上更为简洁。需要注意的是，switch 语句所隐藏的等值比较运算是不会对数据类型进行转换的 === 运算。如果原来的 if-else 语句中的表达式使用的是 == 运算的话，就可能会在执行上有一些细微的差别。而这样的问题通常不容易被发现，很容易成为产生错误的根源。

4.14 循环语句

和条件分支语句一样，循环语句也是基本的控制语句。循环处理语句的一个不太严密的定义是，只要某个条件成立就不断重复执行同样处理的控制语句。由于源代码中同一代码块会被反复执行，所以也称为循环指令处理。

在 JavaScript 中有以下 5 种循环语句。for each in 语句在 ECMAScript 中并不存在，是 JavaScript 独有的增强功能。

```
// while 语句
while （条件表达式）
    语句

// do-while 语句
do
    语句
while （条件表达式）;

// for 语句
for （初始化表达式 ； 条件表达式 ； 更新表达式）
```

```
    语句

// for each in 语句（非 ECMAScript 标准功能）
for each （表达式 in 对象表达式）
    语句
```

4.15 while 语句

while 语句是最为基本的循环控制语句。while 语句也被称为 while 循环。下面是 while 语句的语法规则。

```
// while 语句的语法规则

while （条件表达式）
    语句
```

和 if-else 语句一样，在条件表达式位置所写的表达式的值将会被转换为布尔型。一旦开始执行 while 语句，就将首先对条件表达式求值。如果该值为假，则不会执行循环部分的语句并结束该 while 语句。如果条件表达式的值为真，则将执行该语句。在语句执行结束之后，会再次对条件表达式求值。如果此时该值仍然为真，则再次执行循环部分的语句。在条件表达式的值变为假之前，将一直重复执行循环部分的语句。

和 if-else 语句一样，可以在循环部分的语句处使用代码块（代码清单 4.4）。在本书中，原则上会始终采用代码块的形式书写。

代码清单 4.4　while 语句和代码块

```
// 容易产生混淆的缩进方式。第二个 print 其实是在 while 循环之外的。
while (flag)
    print("in loop");
    print("not in loop"); // 循环之外的语句

// 使用代码块的话就能很容易地理清语句结构
while (flag) {
    print("in loop1");
    print("in loop2");
}

// 即使语句只有 1 行，也推荐使用代码块
while (flag) {
    print("in loop1");
}
```

在 while 语法结构中可以使用任意的语句。if-else 语句也属于是语句，while 语句也属于语句。也就是说，在下面这样的 while 的循环中，也是可以使用 while 语句或者是 if-else 语句的。

```
// 嵌套的 while 语句
while (flag) {
    while (flag2) {
           print("in double loop");
    }
}

// while 语句中嵌入 if 语句
while (flag) {
    if (flag) {
        print("in loop");
    }
}
```

在条件表达式始终为真的情况下，循环内的语句将会被无限次重复执行。这样的循环一般称为无限

循环。意外形成的无限循环是一种致命性的错误。

在其他的程序设计语言中，有时会有意地使用一些无限循环，但在客户端 JavaScript 开发中，无限循环就是一种错误。

要避免出现无限循环，从 while 循环中跳出，可以执行以下操作。

- 保证在循环过程中条件表达式的值将变为假
- 在循环内部使用 break 语句
- 在循环内部使用 return 语句
- 在循环内部抛出异常

break 语句、return 语句以及异常的概念都将在之后的有关跳转的小节中进行说明。

对于在循环过程中条件表达式的值变为假的情况，在此考虑一个进行 10 次循环的 while 循环的例子。其实对于这个例子的功能，用之后将要介绍的 for 循环会更加直观，在这里仅仅因示例的需要而使用了 while 语句。

```
// 循环 10 次的 while 语句
var i = 0;
while (i < 10) {
    print(i);
    i++;
}
```

```
// 循环 10 次的 while 语句（另一种实现方式）
var doing = true;
var i = 0;
while (doing) {
    print(i);
    i++;
    if (i == 10) {
        doing = false;
    }
}
```

第二种方式中通过改变标记变量来跳出 while 循环的做法，并不符合编程习惯。

4.16 do-while 语句

do-while 语句是另一种循环语句，其语法结构如下所示。

```
// do-while 语句的语法结构

do {
    语句
} while （条件表达式）;
```

do-while 语句的条件表达式，和循环内部的语句、while 语句是相同的，所以在此不再重复说明。同样地，本书原则上始终会对循环部分的语句使用代码块。

```
// do-while 语句的例子
do {
    print("in loop");
} while (flag);
```

while 语句与 do-while 语句的差别仅仅在于，是首先对条件表达式进行求值，还是首先执行语句。对于 while 语句，先对条件表达式进行求值，如果表达式的值为真，才执行循环部分的语句。之后将不断

循环直至条件表达式的值变为假。

对于 do-while 语句，则首先执行循环部分的语句，之后才对条件表达式进行求值。之后同样会不断循环直至条件表达式的值变为假。

一旦循环开始，条件表达式的求值与循环内语句的执行就将会交替进行。因此如果非要界定 while 语句和 do-while 语句的差别，那就只有是否会进行最初的那一次条件表达式的求值而已。

在实际编程中，do-while 语句的使用并不多，主要使用 while 语句。其实，do-while 语句的使用模式只有以下两种。只要稍加调整，这两种情况也都能通过 while 语句来实现。

- **如果循环内的语句不执行一次，条件表达式的求值就没有意义的情况**
- **希望确保循环内的语句至少被执行一次的情况**

下面是一个使用 do-while 语句的具体示例。这是一个从右往左逐一显示参数所提供数值的字符的函数。如果输入的内容是 123，则会依次输出 3、2、1。如果不使用 do-while 语句而是使用 while 语句来改写这一函数，当输入 0 时输出就会为空。而使用了 do-while 之后，如果传递的参数是 0，则会输出 0。

```
// 使用 do-while 语句的例子
function printNumberFromRight(n) {
    do {
        print(n % 10);
        n = ~~(n / 10);
// 如果使用 n /= 10; 的话结果将会是一个小数。~~ 运算是一种可以把小数变为整数的巧妙方法。
    } while (n > 0);
}
```

4.17 for 语句

for 语句是另一种循环语句。for 语句一般称为 for 循环，其语法结构如下所示，其中的三个表达式都是可以省略的。

```
// for 语句的语法

for ( 初始化表达式 ; 条件表达式 ; 更新表达式 )
    语句
```

和其他几种控制语句一样，for 语句中的循环部分也推荐使用代码块。

在初始化表达式中，通常会写诸如 i=0 这样的对变量进行初始化的表达式。通常会把在初始化表达式中进行初始化的变量称为循环变量。一个典型的 for 语句，将会在初始化表达式中对循环变量进行初始化，在更新表达式中对循环变量进行更新，在条件表达式中检查循环变量的值。之后会对 for 语句的习惯用法作进一步说明。

初始化表达式只会在 for 语句执行之初被求值 1 次。如果省略了初始化表达式，也只不过是在 for 语句执行时不进行这次求值而已。

在初始化表达式中还可以书写循环变量的声明表达式。需要注意的是，这一变量的作用域和函数作用域是相同的，而非仅限于语句之内。更详细的内容请参见 6.4 节。

```
// 在初始化表达式中对循环变量 i 进行声明的例子
for (var i = 0; i < 10; i++) {
    print(i);
}
注意，这时循环内是可以引用变量 i 的 ( 处于作用域之内 )。
```

条件表达式的作用和 while 语句以及 do-while 语句的基本相同。如果返回值为真，则继续循环；如果为假，则跳出循环。对条件表达式进行求值的时机和 while 语句是相同的，也就是说，for 语句会在执行

之初就对条件表达式进行求值，如果条件表达式的值为真，则执行循环内的语句，在语句执行之后再次对条件表达式进行求值。和 while 语句不同的是，for 语句会在第 2 次对条件表达式进行求值之前，先对更新表达式进行求值。此外，与 while 语句和 do-while 语句不同，for 语句的条件表达式是可以省略的。如果省略了该部分，则会认为条件表达式的值永真。

在更新表达式的位置可以书写任意的表达式。大多数情况下，会在此更新循环变量的值。更新表达式会在 for 循环内的语句执行之后执行，之后条件表达式将被再次求值。之后将会讲到，即使在 for 语句中使用了 continue 语句，也仍然会对更新表达式进行求值。更新表达式也是可以被省略的。如果省略了该部分，就不再执行这一步骤。

for 语句的习惯用法

for 语句的主要用途是，在某个值的范围内从头到尾执行一遍处理。下面是一个最为典型的具体示例。

```
// for 语句为人所熟知的习惯用法
for (var i = 0; i < 10; i++) {
    print(i);
}
```

这段代码是 for 循环的一个很有名的例子，其执行结果是输出从 0 至 9 的 10 个数字。把代码中的 10 替换成其他任意整数 n 的话，就将会循环 n 次。之前已经提到过，使用 while 循环也可以实现同样的功能。不过在习惯了 for 语句的结构之后，会感受到 for 语句有着更高的可读性。

有经验的程序员由于见惯了这样的惯用形式，所以只要粗略读一下代码就马上能判断出这段代码将会循环 10 次。也就是说，在从 0 开始直至小于 10 的范围内，循环将会被执行 10 次。不过，对于经验不足的程序员来说，可能会觉得这样的规则有些别扭。

对于习惯于通常的计数方式的人来说，可能会凭直觉认为要进行 10 次循环的话应该是从 1 开始至 10 结束，于是将会写出下面这样的 for 循环。

```
// 循环变量从 1 开始并且循环了 10 次。如果没有特别的理由，请不要这样书写。
for (var i = 1; i <= 10; i++) {
    print(i);
}
```

如果是需要变量 i 的取值是从 1 至 10，这样的代码是没有问题的。但是，如果是需要代码循环 10 次的情况，还是请使用习惯用法。虽然有读者可能会认为这两种方式都将循环 10 次，选择哪种都没问题，不过因为可能会让其他人产生误解，所以还是请使用习惯用法。程序设计语言不仅仅是开发者向计算机传递意图的语言，也是为了让程序的阅读者能够理解的语言。因此，有必要在此选择使用惯用形式（或者说是一种固定格式）。

初始化表达式和更新表达式还可以像下面这样，通过逗号来分隔多个并列的表达式。

```
// 多个循环变量的例子
for (var i = 0, j = 0; i < 10 && j < 10; i++, j++) {
    省略
}
```

4.18 for in 语句

for in 语句是用于枚举对象属性名的循环语句，其语法结构如下。

```
// for in 语句的语法结构
```

```
for （变量 in 对象表达式）
    语句
```

in 的左侧是在语句中供赋值的表达式，能够在循环时对其进行操作。从语法上来说是左值。所谓左值就是写在赋值表达式左侧的值，只要简单地理解为需要在此写的是一个变量名就足够了。和 for 语句一样，可以在 for in 语句内进行变量声明（同样地，其作用域也不是仅仅在 for in 语句之内）。今后，也将把该变量称为循环变量。

in 的右侧是 Object 类型的表达式。之前提到过，由于在 JavaScript 中会进行隐式的数据类型转换，而任何类型的值都可以被转换为 Object 类型，所以事实上，这里可以书写任意类型的值。因此，无论是数值还是布尔值，都不会引起错误。但是那样做并没有什么实际意义，所以最终还是应当在此书写 Object 类型的表达式。关于 for in 语句具体的执行方式，将会在有关对象的那一节中再进行说明。

在对象表达式处所写的对象的属性名的字符串，将会被依次赋值给循环变量。通过具体的例子会比文字说明更为容易理解，请参见代码清单 4.5。

代码清单 4.5　枚举对象的属性名

```
var obj = { x:1, y:3, z:2 };
for (var k in obj) {
     print(k);
}
```

```
// 代码清单 4.5 的运行结果
x
y
z
```

对象 obj 含有三个属性。通过 for in 语句可以把其属性名为 x、y、z 的字符串赋值给循环变量 k。通常会把对象看作一个关联数组，而把循环变量命名为 k 或是 key，使人联想到这是关联数组的键。有时也把循环变量命名为 p 或 n(ame)，使人联想到这些是属性名称。

如果要显示属性值的话，通常会像代码清单 4.6 这样，在 for in 语句中使用中括号运算。

代码清单 4.6　枚举对象的属性值

```
var obj = { x:1, y:3, z:2 };
for (var k in obj) {
     print(obj[k]);
}
```

```
// 代码清单 4.6 的运行结果
1
3
2
```

4.18.1　数列与 for in 语句

由于数列也是一种对象，且其下标数值相当于一种属性名，所以也可以像下面这样通过 for in 语句来实现枚举。不过，并不推荐采用这种方式对数列的元素进行枚举，之后会再做说明。

代码清单 4.7　枚举数列的元素（不推荐这么做）

```
var arr = [7, 1, 5];
for (var n in arr) {
     print(n + '=>' + arr[n]);
}
```

```
// 代码清单 4.7 的运行结果
```

```
0=>7
1=>1
2=>5
```

4.18.2 在使用 for in 语句时需要注意的地方

关于 for in 语句的使用，这里有 3 点需要注意的。

■ 枚举属性的顺序

首先是枚举属性的顺序。虽然在代码清单 4.5 中，是以对象字面量书写的顺序来进行枚举的，不过实际上并非一定会这样。毕竟属性之间本身就不存在顺序关系，所以认为它们应该以某种顺序排列的想法，从根本上就是不正确的。

另一方面，数列则是一种有顺序关系的数据类型。代码清单 4.7 的输出顺序就和预想的情况相同。虽然可能绝大多数时候都能够像这样获得预期的输出顺序，不过 for in 语句本身并不保证会按照某一顺序枚举，所以仍然不应该对此过分依赖。

■ 无法被枚举的属性

第二个要注意的是，有一些属性是不能够被 for in 语句枚举的。例如，虽然数列对象中含有 length 这一属性，但在代码清单 4.7 中 for in 语句并没有枚举它。这是因为 length 是一种无法被 for in 枚举的属性。更详细的信息请参见 5.10 节。

■ 由原型继承而来的属性

第三个要注意的是，for in 语句还可以枚举由原型继承而来的属性。详细内容请参见 5.8.3 节以及 5.16 节。

4.19 for each in 语句

for each in 语句在 ECMAScript 中并不存在，是 JavaScript 自有的增强功能。for each in 语句的语法结构如下所示。由于变量和对象表达式的部分和 for in 语句相同，所以在此不再赘述。

```
// for each in 语句的语法

for each ( 变量 in 对象表达式 )
    语句
```

和 for in 不同，for each in 语句并不是把属性名赋值给变量，而是将属性值赋值给它。通过一个具体的例子就能很容易地理解了，请将下面的例子和代码清单 4.5 做比较。

```
var obj = { x:1, y:3, z:2};
for each (var v in obj) {
    print(v);
}
```

```
// 运行结果
1
3
2
```

for each in 语句的基本概念和 for in 语句是相同的，关于 for in 语句的那些注意事项对于 for each in 语句也一并通用。

4.20 break 语句

在循环中有时会需要中途跳出，为此可以使用 break 语句。break 语句不仅可以用于跳出 switch-case 语句的循环，也能够用于其他类型的循环语句。下面是不通过 break 语句跳出 while 循环的例子和通过 break 语句跳出循环的例子（代码清单 4.8）。

代码清单 4.8　break 语句的例子

```
// 不通过 break 语句跳出循环的代码示例
var flag_loop = true;
while (flag_loop) {
    省略
    if （跳出循环的条件） {
        flag_loop = false;
    }
}

// 通过 break 语句跳出循环的代码示例
while (true) {
    省略
    if （跳出循环的条件） {
        break;
    }
}
```

严格来说，使用了 break 语句的代码和第一段代码并不是等价的。这是因为使用了 break 语句的情况下，不会再一次对条件表达式进行求值。如果是 for 语句的情况，则不会对更新表达式求值。不过这样的差别通常并不会引起问题。

4.21 continue 语句

在循环中使用 continue 语句的话，就会跳过在此之后本次循环内尚未执行的语句，而返回至循环的条件表达式进行求值。如果是 for 语句的情况，则是返回至更新表达式处求值后再对条件表达式进行求值。更为直接点说，就是 continue 语句将会跳转至循环的开头。

下面的代码是一个仅当循环变量为偶数时才进行处理的 for 循环的例子。

```
// 仅当循环变量为偶数时才进行处理的 for 循环
for (var i = 0; i < 10; i++) {
    if (i % 2 == 0) {
        处理
    }
}
```

这段代码可以改写为在循环的头部对循环变量是否为偶数进行判断，并通过 continue 来进行跳转的形式。使用了 continue 语句之后，实际的处理部分可以减少一层。通常，减少缩进的层数可以提高代码的可读性。

```
// 仅当循环变量为偶数时才进行处理的 for 循环（采用了 continue 的版本）
for (var i = 0; i < 10; i++) {
    if (i % 2 != 0) {
        continue;
    }
    处理
}
```

4.22 通过标签跳转

在嵌套循环的内层使用 break 语句的话，仅仅会跳出内层的循环。请看下面的例子。

```
// 在嵌套循环中使用 break 语句的代码示例。仅仅会跳出内层循环。
while （条件表达式） {
    print("outer loop");
    while （条件表达式） {
       print("inner loop");
       if （跳出循环的条件） {
          break;
       }
    }
}
```

那么该如何才能同时跳出外层循环呢？从理论上来说，可以像下面这样通过旗标变量来实现。不过，这样的代码不容易改动，所以并不推荐。

```
// 同时跳出嵌套内外层的循环（采用了旗标变量的版本，并不推荐）
var flag_loop = true;
while (flag_loop) {
    print("outer loop");
    while （条件表达式） {
       print("inner loop");
       if （跳出循环的条件） {
          flag_loop = false;        // 这样就能同时跳出外层循环
          break;
       }
    }
    flag_loop 变量变为了 false 之后，将会跳过这一部分代码
}
```

还可以使用标签实现同时跳出嵌套的循环。标签的语法规则如下。

```
// 标签的语法规则

标签字符串 ： 语句
```

在标签字符串处可以使用任意的标识符。标签所指向的语句可以是任意类型的，并不一定必须是循环语句，不过在大多数时候，标签会指向循环语句。

可以像下面的代码这样，使用标签来实现同时跳出内外层嵌套的循环。

```
// 使用标签来同时跳出嵌套的循环
outer_loop:
while (true) {
    print("outer loop");
    while (true) {
       print("inner loop");
       break outer_loop;
    }
}
```

这段代码可以这样解读：外层循环被标以 outer_loop 的标签（此前提到过，请再回想一下，while 循环以及相应的代码块共同组成了一句语句）。通过 break outer_loop 就可以跳出标有该标签的语句。这样就能一下子从嵌套的循环中跳出了。

对 continue 语句也可以使用标签。这时，将会跳转至标有相应标签的循环语句的条件表达式处进行求值（对于 for 语句来说则是相应的更新表达式与条件表达式）。

4.23 return 语句

return 语句的语法如下所示，其中表达式是可以省略的。

```
// return 语句的语法结构

return 表达式 ;
```

return 语句会中断函数的处理，并将指定的表达式的值作为函数的返回值返回。如果没有指定表达式，函数的返回值将会是 undefined 值。

4.24 异常

可以通过 throw 语句来抛出异常对象（异常值）。throw 语句的语法规则如下所示。

```
// throw 语句的语法规则

throw 表达式 ;
```

其中表达式处可以使用任意类型的表达式，这与只能够使用异常类型的 Java 是不同的。

在捕捉异常的地方，则需要使用 try-catch-finally 的结构。其中 catch 子句和 finally 子句是不能同时省略。如果只省略其中一个则没有问题。

```
// try-catch-finally 结构的语法

try {
    语句
    语句
    ……
} catch （变量名） { // 该变量是一个引用了所捕捉到的异常对象的局部变量
    语句
    语句
    ……
} finally {
    语句
    语句
    ……
}
```

如果在 try 子句中（以及在 try 子句中调用的函数内）发生异常的话，运行就会中断，并开始执行 catch 子句的部分。执行 catch 子句被称为捕捉到了异常。在 try 语句之外，或者没有 catch 子句的 try 语句，都是无法捕捉异常的。这时函数会中断并返回至调用该函数之处。

throw 语句和 return 语句在中断函数的执行上是相似的，不过 throw 语句并没有返回值。而且，如果没能在调用了该函数的函数内的 catch 子句中捕捉异常的话，还会进一步返回到更上一层的调用函数（这种行为称为异常的传递）。

如果最终都没能成功捕捉异常，整个程序的执行就将被中断。

finally 子句必定会在跳出 try 语句之时被执行。即使没有产生异常，finally 子句也会被执行。也就是说，如果没有产生异常的话，在执行完 try 子句之后会继续执行 finally 子句的代码；如果产生了异常，则

会在执行 finally 子句之前首先执行 catch 子句。对于没有 catch 子句的 try 语句来说，异常将会被直接传递至上一层，但 finally 子句仍然会被执行。

异常可以通过 throw 语句显式地抛出，也可能在特定的运算中产生。代码清单 4.9 中是一段仅仅显示了执行顺序而没有其他实际意义的代码。

代码清单 4.9 try 语句的执行示例

```
try {
    print('1');
    null.x;                  // 在此处强制产生一个 TypeError 异常
    print('not here');       // 这条语句不会被执行
} catch (e) {                // 对象 e 是 TypeError 对象的一个引用
    print('2');
} finally {
    print('3');
}
```

```
// 代码清单 4.9 的运行结果
1
2
3
```

JavaScript 和 Java 在 try 语句上最大的区别是异常类型的不同。在 Java 中可以对异常进行类型分类，选择多个 catch 子句中的某一个来捕捉异常。但是在 JavaScript 中，catch 子句不能根据异常的类型不同而决定是否捕捉该异常。

毕竟，在 JavaScript 中一个 try 语句只能使用一个 catch 子句，因此这个 catch 子句可以捕捉任意类型的异常。要把异常传递至上一层（函数的调用方），就不能使用 catch 子句，或者需要在 catch 子句内重新抛出异常对象。

4.25 其他

with 语句的语法结构如下所示。

```
// with 语句的语法结构

with （表达式）
    语句
```

with 语句用于临时改变名称（变量名或是函数名）的查找范围。with 语句中使用的表达式是 Object 类型的。如果使用了其他类型的值，则会被转换为 Object 类型。在 with 语句内对变量名进行查找时，将会从所指定对象的属性开始寻找。

下面是一个具体的例子。

```
// with 语句的例子

js> var x = 1;          // 全局变量
js> var obj = { x:7, y:8 };
js> with (obj) {
    print(x);           // 如果要查找变量 x，则会在查找全局变量 x 之前先查找到 obj.x
    }
7
```

然而，在 ECMAScript 第 5 版的 strict 模式中是禁止使用 with 语句的。因此本书也将遵循这一方针，不使用 with 语句。

debugger 语句是在 ECMAScript 第 5 版中出现的语句，顾名思义，它用于调试。其具体作用为，调试过程将会在含有 debugger 语句的代码行中断。

4.26 注释

注释分为以下两种类型。

```
// 单行注释
/* 注释 */
```

4.27 表达式

要对表达式（expression）有一个直观的理解，可以从计算表达式入手。例如，1+1 在数学中是一种表达式，在 JavaScript 中也是一种表达式。1+1 是由数字 1 和表示加法运算的 + 运算符组成的。作为运算对象的数字被称为操作数。运算符和操作数在英语中分别称为 operator 和 operand。

由于程序设计语言中可以进行运算的对象不仅仅有数值类型，所以通常会将操作数直接称为 operand。虽然日语和英语在形式上并不对仗，但本书中还是会使用运算符和 operand 这样的术语①。

通过这些术语，就能够直接说明表达式的含义了，即由运算符和操作数连接而成的式子。运算符在 JavaScript 的语言标准中有着明确的定义。每一个运算符的含义与功能都将会在各小节中进行说明。

4.28 运算符

表 4.5 对 JavaScript 中的运算符进行了总结。而关于优先级的问题则将在之后再进行详细说明。

表 4.5 JavaScript 的运算符（按优先级顺序排列）

.	[]	New									
()											
++	--										
!	~	+（单目）	-（单目）	typeof	void	delete					
%	*	/									
+（双目）	-（双目）										
<<	>>	>>>									
<	<=	>	>=								
==	!==	===									
&											
^											
|											
&&											
||											
?:（三目）											
=	+=	-=	*=	/=	%=	<<=	>>=	>>>=	&=	^=	|=
,											

对于每一个运算符，其操作数的数量与位置，或者可以使用哪些类型的操作数等都是有规定的。例如加法运算符 +，其操作数的数量是 2。有两个操作数的运算符被称为双目运算符，需要在运算符的前后

① 这里是一处与日文译法有关的说明。对于中文来说则不存在这一问题，故中文版将仍然使用“操作数”这一术语。——译者注

分别书写一个操作数。大部分运算符都是双目运算符，三目运算符有 1 个，其余的都是单目运算符。三目运算符有 3 个操作数，而单目运算符则只有 1 个操作数。单目运算符又可以根据运算符与操作数的前后位置关系，分为前置运算符与后置运算符。前置运算符是以"运算符 操作数"的顺序书写的，而后置运算符则是按照"操作数 运算符"的顺序。

对于某一特定的运算符来说，这些规则都是固定不变的。在单目运算符中，有一些既可以作为前置运算符也可以作为后置运算符（例如 ++ 运算符等）。这里并不是说这些运算符把操作数写在前面或是后面都没有关系，而是说这些运算符在前置与后置使用的情况下都有其特定含义，所以请将其理解为，不同的运算功能恰巧采用了相同的符号表示。

4.29 表达式求值

根据习惯，对于语句来说，采用的是执行（execute）这个词，而对于表达式来说，则会采用求值（evaluate）这个词。表达式求值的基本过程是先对操作数（或所对应的表达式）求值，之后再应用表达式中的运算符。不过有一部分运算符例外（逻辑运算符 ||、逻辑运算符 &&，以及三目运算符），它们会在最后的阶段才对操作数求值，甚至有时不会对操作数进行求值。具体情况需要具体分析。

对于双目运算符的情况，对左操作数的求值将先于右操作数。对于三目运算符的情况，首先会对最左侧的操作数求值，之后根据具体情况只会对剩余两个操作数中的一个进行求值。

从表达式中哪个位置开始求值是由运算符的优先级决定的。之后会再详述有关优先级的内容。在对表达式求值之后，就能够得到求值结果，该结果称为表达式的值。

以下是对表达式求值中优先级问题的总结。

- 除了含有 && 运算符、|| 运算符、?: 运算符这三个运算符的情况外，其他的表达式都是首先对操作数进行求值。
- 操作数按从左至右的顺序求值。
- 对于函数方法或是构造函数调用表达式的情况，会在调用前对参数从左至右求值。
- 优先对括号内的表达式求值。
- 如果在对操作数求值的过程中产生了异常，则不会对剩余的操作数进行求值。
- 对于函数方法或是构造函数调用表达式的情况，如果在对参数进行求值的过程中发生异常，则不会对剩余的参数进行求值。

JavaScript 中的运算规则其实和 Java 并没有太大的差别。不过，由于在 JavaScript 中存在很多隐式数据类型转换，所以有时不太容易理解运算的实际行为。例如，对字符串值和数值进行比较的情况，在 Java 中会发生编译错误，而在 JavaScript 中则会先进行隐式数据类型转换，之后正常比较。不会发生编译错误貌似是一种优点，但实际上这会导致更高的复杂性。隐式数据类型转换一方面为开发提供了便利，另一方面也常常是产生错误的原因。

4.30 运算符的优先级以及结合律

运算符之间有着优先级（参见表 4.5）。例如，在下面的代码中，乘法运算会先于加法运算。此外，和通常的算术表达式一样，在 JavaScript 中也可以通过添加括号来改变运算顺序。

```
// 运算符的优先级
js> 1 + 2 * 3;            // 将会首先计算 2*3
7
```

对于优先级相同的运算符，则是根据运算符的结合规则来决定运算的先后顺序。结合规则分为左结合与右结合两种。+ 运算符是左结合的，所以代码清单 4.10 中表达式里的第一个 + 运算符（1 与 2 之间的那个）的左操作数是 1，右操作数是 2。第二个 + 运算符（2 与 3 之间的那个）的左操作数是 1+2 的值，右操作数是 3。

假如 + 运算符是右结合的话，情况将变为第一个 + 运算符的左操作数为 1，右操作数为 2+3（的值），第二个 + 运算符的左操作数为 2，右操作数为 3。

代码清单 4.10　算术运算符的结合律为左结合

```
1 + 2 + 3
将以
(1 + 2) + 3
的方式被求值
```

- **前置单目运算符是右结合的。**
- **后置单目运算符是左结合的。**
- **除赋值运算符之外的双目运算符都是左结合的。**
- **赋值运算符是右结合的。**
- **三目运算符是右结合的。**

单目运算符的结合律是显而易见的。对于除了赋值运算符以外的双目运算符，以算术运算符的方式来考虑的话也不难理解。而三目运算符本身就是比较特殊的运算符，需要另外考虑。于是，结合律比较特殊的其实就只有赋值运算符而已。之所以在双目运算符中只有赋值运算符是右结合，是为了应对下面这种同时对多个变量进行赋值的赋值表达式的情况。

```
// 赋值表达式的结合律为右结合
x = y = z = 0;
将以
x = (y = (z = 0);
的方式被求值
```

4.31 算术运算符

表 4.6 对算术运算符进行了总结。如果操作数不是数值，则会先将其转换为数值后再进行计算。其运算结果也是数值。在之后还会对 + 运算符的一些注意事项进行说明。

表 4.6　算术运算符

运算符	说明
+	加法
-	减法
*	乘法
/	除法
%	取模
++（前置）	自增
++（后置）	自增
--（前置）	自减
--（后置）	自减
-（单目）	符号反转
+（单目）	符号保持不变

对于算术运算符，有一些需要注意的地方。首先，+ 运算符比起加法运算符，会优先作为字符串连接运算符并对操作数进行数据类型转换。详细信息请参见 3.9.1 节。

然后需要注意的是，在 JavaScript 所有算术运算中的数值都是浮点小数。1/2 的结果将会是 0.5。如果只考虑 JavaScript 一种语言可能会认为这是理所应当的，不过因为在那些具有整型数值的语言中 1/2 的结果为 0，所以这一点对于习惯了其他语言的人来说，反而会是一个容易出错之处。此外，在 JavaScript 中一个数被 0 除之后也不会发生错误，而是得到 NaN 的结果。

虽说算术运算符会把操作数转换为数值类型，不过如果操作数比较特殊的话，转换后的结果也可能会是 NaN。操作数是 NaN 的算术运算结果始终是 NaN。在大部分情况下这将引起错误，无法得到期望的结果。

```
js> 'x' * 0;
NaN
js> +'x';
NaN
```

++ 运算符的含义是对操作数加 1。-- 运算符的含义是对操作数减 1。这两个运算符都会重写操作数的值。这类会改写操作数本身的值的运算符，称为破坏性运算符[①]。因此，需要在左值处书写操作数，也就是说在赋值表达式的左边书写表达式。这么说可能有些难以理解，总之 ++ 运算符与 -- 运算符的操作数将作为右值读取，并且在其左值处需要书写表达式。

前置运算符与后置运算符的区别在于运算结果的值不同。前置运算符的值，是进行了加法或减法运算之后的值。而后置运算符的值，则是在进行加法或是减法运算之前的值。请看具体的例子（代码清单 4.11）。

代码清单 4.11　前置运算符与后置运算符的区别

```
// 前置运算符的行为
var n = 10;
var m = ++n;

print(m, n); // n 变为了 11。++n 的值是进行了加法之后的值，所以 m 为 11。
11 11

// 后置运算符的行为
var n = 10;
var m = n++;

print(m, n); // n 变为了 11。++n 的值是进行了加法之前的值，所以 m 为 10。
10 11
```

之前已经说过，++ 运算符、-- 运算符是破坏性运算符。因此如果不注意 JavaScript 中的隐式数据类型转换的话，就很容易产生错误（请参见 3.9.7 节）。

4.32 字符串连接运算符

+ 运算符和 += 运算符是两个字符串连接运算符，其运算结果为字符串值。如果操作数不是字符串型，则会先对其进行数据类型转换之后再进行运算。

此类运算符的详细信息请参见 3.3.2 节。

4.33 相等运算符

在 JavaScript 中有两种相等运算符，即 === 和 ==，它们分别有对应的不相等运算符 !== 和 !=。在

① 赋值运算符是另一种有代表性的破坏性运算符。

ECMAScript 中，=== 被称为 Strict Equals 运算符，而 == 则被称为 Equals 运算符。

这两个运算符有好几种翻译方式。在本书中，将 Strict Equals 运算符（===）称为全等运算符，而将 Equals 运算符（==）称为相等运算符。两者的区别在于，是否会在进行相等判定时进行数据类型转换。全等运算不会进行数据类型转换，因此数据类型是否一致也是判断是否相等的内容之一。而相等运算（==）会先进行数据类型转换，在数据类型相同后再进行相等判断。两种运算符的运算结果都是布尔值。

全等运算与它给人的第一印象较为相近。除了对字符串的比较是比较其内容外，其执行方式和 C 语言或是 Java 之类的静态程序设计语言中的相等运算（==）几乎是一样的。不过，对于那些静态程序设计语言来说，如果两侧的数据类型不同，通常会导致编译错误（有些则会进行数据类型转换后再做相等运算）。而在 JavaScript 中，全等运算对于这种情况仅仅是得到一个为假的结果而已。下面总结了全等运算的一些特性。

1. x 与 y 如果数据类型不相符，则结果为假。
2. 两者都是 undefined 值或两者都是 null 值的情况，结果为真。
3. 两者都是数值，但有一方为 NaN，或者两者都是 NaN 的情况，结果为假。否则，如果数值相等则结果为真，不相等则为假。
4. 两者都是字符串的情况下，如果内容一致则结果为真，否则结果为假。
5. 两者都是布尔值的情况下，如果值一致则结果为真，否则结果为假。
6. 两者都是对象引用的情况下，如果引用的是同一个对象则结果为真，否则结果为假。

相等运算 == 由于会进行隐式数据类型转换，所以其执行方式更为复杂。下面是对其运算规则的总结。

- x 与 y 的数据类型相同时，与全等运算的结果相同。
- x 与 y 的数据类型不同时，判定规则如下。

（1）一方为 null 值，另一方为 undefined 值的情况，结果为真。
（2）一方为数值，另一方为字符串值的情况，将字符串值转换为数值之后对数值进行比较。
（3）一方为布尔值，另一方为数值的情况，将布尔值转换为数值之后对数值进行比较。
（4）一方为布尔值，另一方为字符串值的情况，将两者都转换为数值后对数值进行比较。
（5）一方为数值，另一方为对象引用的情况，将对象引用转换为数值后对数值进行比较。
（6）一方为字符串值，另一方为对象引用的情况，将对象引用转换为字符串值后对字符串的内容进行比较。
（7）以上 6 种情况之外的运算结果都为假。

4.34 比较运算符

>、<、<= 和 >= 这四种运算符是比较运算符。比较运算符的运算结果为布尔值。如果两个操作数都是数值，则对这两个数值的大小进行比较。如果两个操作数都是字符串型则对字符串值内容的 Unicode 编码进行大小比较。

如果两个操作数的类型不同，则遵循以下规则。

1. 一方为数值，另一方为可以被转换为数值的数据类型的情况，将其转换为数值类型后再进行大小比较。
2. 如果操作数中含有 NaN 则结果为假[①]。
3. 一方为字符串值，另一方为可以被转换为字符串值的数据类型的情况，将其转换为字符串值后再对字符串值进行大小比较。
4. 操作数中有无法被转换为数值及字符串值的值，或是转换结果为 NaN 的情况，运算的结果为假。

① 在 ECMAScript 第 5 版中其运算结果为 undefined 值。

在两个操作数都是字符串型的情况下，将对字符串进行大小比较。但如果操作数中只有一个是字符串的话，则会将该字符串转换为数值类型。请看下面的例子。

```
js> '100' > '99';            // 以字符串值形式比较（由于 '9' 的编码值大于 '1'，因此结果为 false）
false
js> '100' > 99;              // 字符串值将转换为数值，判断 100>99 并得到结果 true
true
```

在通常的数学概念中，对 x 与 y 两个数值进行大小比较时，x>y 和 x<=y 中只会有一方的结果为真，不会发生两者都为真或是两者都为假的情况。不过在 JavaScript 中，比较运算可能会经过数据类型转换，所以可能出现两者都为假的情况。这可能会引起错误，所以有必要加以注意。下面是一个具体的例子。

在此仅对 > 与 <= 的情况作了说明，> 与 <= 的情况类似。

```
// 转换为数值类型后变为了 NaN 的字符串值（参见表 3.17）
js> 1 > 'x';                 // >= 的结果也是 false
false
js> 1 <= 'x';                // < 的结果也是 false
false

// undefined 值的数据类型转换（参见表 3.18）
js> undefined > 0;           // 在转换为数值之后比较 NaN > 0
false
js> undefined <= 0;          // 在转换为数值之后比较 NaN <= 0
false
js> undefined > undefined;   // 在转换为数值之后比较 NaN > NaN
false
```

下面是一些其他类型的数据转换与大小比较的示例。

```
// 布尔值的数据类型转换（参见表 3.18）
js> true > false;            // 在转换为数值之后比较 1 > 0
true
js> true > 0;                // 在转换为数值之后比较 1 > 0
true

// null 值的数据类型转换（参见表 3.18）
js> null < 1;                // 在转换为数值之后比较 0 < 1
true
js> null > 1;                // 在转换为数值之后比较 0 > 1
false

// Object 类型的数据类型转换（参见表 3.19）
js> var obj = {};
js> obj > 0;                 // 在转换为数值之后比较 NaN > 0
false
js> obj <= 0;                // 在转换为数值之后比较 NaN <= 0
false

js> obj.valueOf = function() { return 1; }
js> obj > 0;                 // 在转换为数值之后比较 1 > 0
true
```

4.35 in 运算符

in 是一种用于检验属性是否存在的运算符，其运算结果为布尔值。具体的执行方式请参见 5.9 节。

4.36 instanceof 运算符

instanceof 是一种用于类型判断的运算符，其运算结果为布尔值。具体的执行方式请参见 5.17.3 节。

4.37 逻辑运算符

表 4.7 对逻辑运算符做了总结。尽管在其他很多程序设计语言中，逻辑运算的运算结果是布尔值，但在 JavaScript 中并没有这样的限制，也可以是非布尔值的结果。不过，由于这可能会使习惯于其他程序设计语言的开发者产生误解，所以并不推荐在代码中使用这一特性。

表 4.7　逻辑运算符

运算符	说明	短路规则
!	逻辑非（NOT）	无（单目运算符）
&&	逻辑与（AND）	若左操作数的值为假，则不再对右操作数进行求值。
||	逻辑或（OR）	若左操作数的值为真，则不再对右操作数进行求值。

如果逻辑运算的操作数不是布尔值，则在将其转换为布尔型之后再进行运算。关于布尔型的转换规则，请参考 3.9.4 节。

在将操作数转换为布尔型之后，! 运算会再对其进行逻辑非运算。逻辑非运算会在调换 true 和 false 值之后返回一个布尔值的结果。

&& 运算和 || 运算有一个重要的性质——短路求值。一般的运算符会在运算前先对操作数进行求值。如果是双目运算符，则会对前后两个操作数事先进行求值后再进行运算。与之不同的是，逻辑运算符在运算前仅会先对左边的操作数进行求值。

只要满足了表 4.7 中的短路规则，运算就会终止，从而不再对右操作数进行求值。此时，运算结果就是左操作数的值（在进行数据类型转换之前的值）。如果没有满足短路规则，则会对右操作数进行求值。运算结果为右操作数的值（在进行数据类型转换之前的值）。由于逻辑运算的结果，是在进行数据类型转换之前的值，所以并不一定是布尔值。下面是一个具体的例子。

```
js> var n = 0 && 1;            // 求值结果为左操作数的值（短路求值）
js> print(n);
0

js> var n = true && 1;         // 求值结果为右操作数的数值
js> print(n);
1

js> var n = 1 || 2;            // 求值结果为左操作数的值（短路求值）
js> print(n);
1

js> var n = false || 'x';      // 求值结果为左操作数的字符串值
js> print(n);
x
```

短路求值是一种条件分支。短路求值仅会在某些条件成立的情况下进行求值，如果不成立则不进行求值。如果使用 if 语句这样的条件分支语句，代码结构会更加清晰。而通过表达式进行条件分支处理的话，则能够使代码变得更为简短。因此在希望代码尽可能短的客户端 JavaScript 开发中，短路求值是一种常见的方式。

有关逻辑表达式中条件分支的习惯用法，请参见 5.5 节。

4.38 位运算符

表 4.8 对位运算符进行了总结。如果操作数不是数值的话，则会先将其转换为内部结构为 32 位整数的数值类型，之后再进行运算。位运算符的运算结果为数值。操作数被转换为整数之后，将其左移 1 位就能使值加倍，将其右移 1 位就能使值减半。由于这与通常的位运算效果相同，所以应该不难理解。

不过，恐怕也不会有读者希望以浮点小数的位直接进行位运算吧，毕竟，即使对浮点小数的位进行左移操作，也不会使其值加倍。

表 4.8 位运算符

运算符	说明
&	按位与（AND）
¦	按位或（OR）
ˆ	按位异或（XOR）
<<	左移
>>	右移（最左位保持原符号不变）
>>>	无符号右移（最左位被置为 0）
~	单目运算符。按位取反，取 1 的补码

根据图 4.1 对每一位进行求值计算之后，就能够得到位运算的结果。

图 4.1 位运算的真值表以及 JavaScript 运算符

<table>
<tr><td>x1</td><td>x2|0</td><td>~(x1|x2)</td><td>–</td><td>~x1</td><td>–</td><td>~x2</td><td>x1^x2</td><td>~(x1&x2)</td><td>x1&x2</td><td>~(x1^x2)</td><td>x2</td><td>–</td><td>x1</td><td>–</td><td>x1|x2</td><td>~0</td></tr>
<tr><td>0</td><td>0 | 0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>0</td><td>1 | 0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0 | 0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>1 | 0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr>
</table>

4.39 赋值运算符

用于对变量赋值的 = 运算符是赋值运算符。由于很多时候赋值表达式都以表达式语句的形式出现，所以很容易将赋值看作一种语句。不过从语法上来说，赋值是一种表达式。下面的例子说明了赋值运算符的右结合特性，从这一点也可以看出这是一种表达式。

```
x = y = z = 0;
```

赋值表达式持有其运算的结果，即所赋的值。上面的 z=0 这一赋值表达式的运算结果为 0。而这一结果又被用于 y=0 这一赋值表达式。然后，0 这一结果又再次被用于 x=0 这一赋值表达式中。这样一来，在 1 个表达式中就实现了对 3 个变量的赋值。

因为赋值是一种表达式，所以下面的代码并没有违反语法规则（事实上，这可以说是一个颇为有名的例子）。

```
// 是否有问题呢?
if (x = y) { 省略 }
```

在这段代码中，表达式 x=y 的结果会被作为 if 语句的条件来进行判定。由于 x=y 的结果为 y 的值，所以这段代码实际上与下面的代码是等价的。y 的值将会被转换为布尔型，若该值为真，则将执行 if 语句中的代码。

```
// 尽管与上一段代码等价，不过这样的程序是否能够按照预期设想的那样工作呢?
x = y;
```

```
if (y) { 省略 }
```

前面之所以说那段代码有名，是因为在大多数时候，它都只不过是写错了以下代码而得到的结果罢了。

```
// 第一段代码大概是希望写成这样的吧
if (x == y);
```

这种错误可以说在整个软件发展史中频繁出现，所以即使在语法上没有问题，也不推荐将赋值表达式作为条件表达式来使用。

4.40 算术赋值运算符

算术赋值运算符指的是表 4.5 中的 +=、-= 等将算术运算符与 = 相连的运算符。算术赋值运算符的作用是将算术运算的结果赋值，相应的表达式运算结果即为该运算结果的值。

4.41 条件运算符（三目运算符）

条件运算符是唯一的三目运算符。由于三目运算符只有这一个，所以有时也会直接把条件运算符称为三目运算符。与 && 运算符和 || 运算符一样，条件运算符也有短路求值的特性。

条件运算符的语法结构如下所示。

```
// 条件运算表达式的语法
条件表达式 ? 表达式1 : 表达式2
```

条件表达式的操作数会首先被求值。得到的值将会转换为布尔型，如果为真的话则对表达式 1 进行求值，如果为假则对表达式 2 进行求值。表达式 1 与表达式 2 中仅有一个会被求值。条件运算表达式的运算结果会是表达式 1 或表达式 2 的值，因此运算结果的类型是由操作数来决定的。

4.42 typeof 运算符

typeof 是一种用于数据类型判定的单目运算符，它支持任意类型的操作数。其运算结果是标识操作数的数据类型的字符串值。在第 3 章中有过介绍，这里通过表 4.9 再次对此进行总结。

表 4.9　typeof 运算

数据类型	typeof 运算的结果
字符串型	"string"
数值型	"number"
布尔型	"boolean"
null 型	"object"
undefined 型	"undefined"
Object 型	"object"
函数	"function"
XML（E4X）	"xml"

对于标准的对象来说，typeof 运算的结果一定是 "object"，不过对于非标准对象，结果则取决于其具体实现。

4.43 new 运算符

new 是用于生成对象的单目运算符。关于 new 运算符的详细信息，请参见 5.7.2 节。

4.44 delete 运算符

delete 是用于删除属性的单目运算符，其功能为从对象中删除以操作数指定的属性。delete 运算的运算结果为布尔值。如果属性被删除，或所要删除的属性不存在，则结果为真，否则结果为假。

其具体执行方式请参见 5.9 节。

在 JavaScript 中，从内部结构来看，全局变量是全局对象的属性（请参见 5.3 节）。不过如果 delete 运算的操作数是一个全局变量，情况则有些特殊。

通过 var 声明的全局变量是无法被 delete 的，而没用使用 var 声明的隐式的全局变量则可以被 delete。正如在第 2 章最初所说的，不推荐使用隐式的全局变量。因此，原则上不应对全局变量进行 delete 运算。

4.45 void 运算符

void 是 undefined 类型的单目运算符。无论向其传递什么操作数，其运算结果都会是 undefined 值。下面是一个具体的例子。

```
js> print(void 0);              // 操作数为数值
undefined
js> print(void 'x');            // 操作数为字符串值
undefined

js> var x = 0;
js> void x++;                   // 由于会先对操作数进行求值，所以 x 将自增
js> print(x);
1

js> void(x);                    // 常常会把操作数通过括号包围起来
```

void 这一运算符的用途比较难以理解，不过在客户端 JavaScript 中有不少相关的习惯用法。下面是一个在 HTML 中点击了标签 a 之后发送表单内容的 JavaScript 代码的例子。

```
<a href="javascript:void(document.form.submit())">发送 HTML 表单数据但不跳转页面 </a>
```

hred 属性中所写的表达式如果具有值的话，则会被标签 a 认为是 URL 并跳转至该页面。为了阻止标签 a 的这一行为，需要将 href 属性中表达式的值强制设为 undefined 值。对此最为简单的惯用方法就是通过 void 运算来实现。

4.46 逗号（,）运算符

逗号运算符（,）是一个双目运算符，其作用为依次对其左操作数与右操作数求值。逗号运算符的运算结果是其右操作数的值，也就是说其结果的类型取决于所使用的操作数。下面是一个具体的例子。

```
js> print((x = 1, y = 2));      // 请注意，如果不在真个参数外加括号的话，其含义就会变为参数的数量是两个
2
js> print((x = 1, ++x, ++x));   // 由于是左结合，相当于((x = 1, ++x), ++x)
3
```

逗号运算符主要用于 for 语句这类需要在表达式位置书写多个表达式的情况。

4.47 点运算符和中括号运算符

字符 .（点）称为点运算符，中括号 [] 称为中括号运算符，它们都是用于访问属性的运算符。虽然这两个运算符不太显眼，却有着很重要的作用。

其左操作数为对象引用，右操作数为属性名。如果左操作数不是对象引用的话，则会被转换为 Object 类型。点运算符的右操作数是一个用于表示属性名的标识符，而中括号运算符的右操作数为字符串型或是可以被转换为字符串型的值。

关于这两个运算符的执行方式，请参见 5.8 节。关于两者在使用上的区别，请参见 5.8.2 节。

4.48 函数调用运算符

函数调用运算符通过 () 来实现对函数的调用，其左操作数是一个函数，而右操作数则是要传递给函数的参数（实参）。对于函数调用运算符究竟有几个操作数，有几种不同的看法，在本书中，将所有的参数看作 1 个操作数，认为函数调用运算符是一个双目运算符。实参将在函数调用之前被求值。

有关函数的详细信息请参见第 6 章。

4.49 运算符使用以及数据类型转换中需要注意的地方

表 4.10 对在进行数据类型转换时需要注意的运算符做了总结。

表 4.10　在进行数据类型转换时需要注意的运算符

运算符	说明
+ 运算符	字符串连接运算优先于加法运算。如果操作数中一方是字符串值而另一方是数值，数值将被转换为字符串值，而后进行字符串连接运算
比较运算符（<、>、<=、>=）	数值比较优先于字符串比较。如果操作数中一方是字符串值而另一方是数值，字符串值将被转换为数值，而后进行数值比较

第 5 章 变量与对象

JavaScript 是面向对象程序设计语言，可以很轻松地通过简洁的字面量形式，以及动态数据类型特性来操作对象。不过需要注意的是，其内部的实现原理与现有其他主流的面向对象程序设计语言是不同的。

5.1 变量的声明

变量的功能为持有某个值，或者用来表示某个对象。关于变量的语法结构，请参见 4.7 节。关于不通过声明而直接使用变量的方式，请参见 2.3.2 节。原则上，本书在使用变量之前一定会对其进行声明。

如果一个变量在声明之后没有进行赋值，它的值就会是 undefined。对同一个变量重复进行声明是不会引起什么问题的，原有的值也不会被清空。请看下面的例子：

```
js> var a = 7;
js> print(a);
7
js> var a;                    // 即使对同一个变量重复进行声明
js> print(a);                 // 它的值也不会发生改变
7
```

上面的代码并没有实际作用，也没有明确意义，所以并不推荐。而下面的代码倒是常常被作为一种习惯用法来使用。

```
var a = a || 7;         // 一种习惯用法。如果变量 a 已经具有某个值（严格来说是具有某个可以被转换为
                           true 的值）就直接使用，否则就把 7 赋值给 a
```

在这段代码中，如果 a 是一个已经被声明且赋值的变量，则不会有任何效果；而如果没有被声明过，则会在声明的同时对其进行赋值操作。

下面的代码虽然和上一段有些相像，却是有问题的。如果变量 b 没有被声明过，将会引起 ReferenceError 异常。不过，也不能说它绝对就是错的。这是因为，如果能确保在这条代码之前就已经对变量 b 进行了声明，这段代码的作用就变为了判定变量 b 的值的真假，这样就没有问题了。

```
var a = b || 7;         // 可能会引起 ReferenceError 异常的危险的代码
```

关于如何判定变量是否已经被声明，请参见之后的 5.5 节。

5.2 变量与引用

对象的概念很好地说明了变量是一种拥有名称的客体。对象本身是没有名称的，之所以使用变量，是为了通过某个名称来称呼这样一种不具有名称的对象。下面是一个在 JavaScript 中将一个对象（的引用）赋值给某一变量的例子。代码中使用了一个空的对象进行赋值。

```
var foo = {};     // 将对象赋值给变量 foo
```

变量又分为基本类型的变量（值型变量）与引用类型的变量。由于在 JavaScript 中，变量是不具有类型的，因此从语法标准上来看，两者并没有什么不同。 不过，在 JavaScript 中仍然有对象的引用这一概念。

所谓“引用”，可以认为是一种用于指示出对象的位置的标记。如果你熟悉 C 语言，把它理解为是和指针等价的东西也没有问题。不过，引用不支持那些可以对指针进行的运算。引用这一语言功能只有指示位置信息的作用。准确地说，对象的赋值其实是将对象的引用进行赋值。

为了更好地解释引用这一概念，这里对引用类型的变量和值型变量进行比较。将基本类型的值赋值给变量的话，变量将把这个值本身保存起来。这时，可以将变量简单地理解为一个装了该值的箱子。变量本身装有所赋的这个值，所以能够将该值从变量中取出。如果在右侧写上一个变量，这一变量的值将被复制给赋值目标处（左侧）的变量。

```
js> var a = 123;            // 将数值 123 赋值给变量 a
js> var b = a;              // 将变量 a 的值（数值 123）赋值给变量 b
```

像下面这样，对变量 b 进行自增操作后，变量 a 的值是不会发生改变的。图 5.1 对这一执行方式作了说明。

```
// 接上面的代码
js> b++;
js> print(b);               // 将 b 的值自增
124
js> print(a);               // a 的值不会发生变化
123
```

另一方面，如果将一个对象赋值给变量，其实是把这个对象的引用赋值给了该变量。对象本身是无法赋值给一个变量的。如果在右侧写上了这样的变量，该变量所表示的引用将被复制给赋值目标处（左侧）的变量。对象本身并不会被复制。

```
js> var a = { x:1, y:2 };  // 将对象的引用赋值给变量 a
js> var b = a;              // 将变量 a 的值（对象的引用）赋值给变量 b
```

图 5.1　值型变量的执行方式

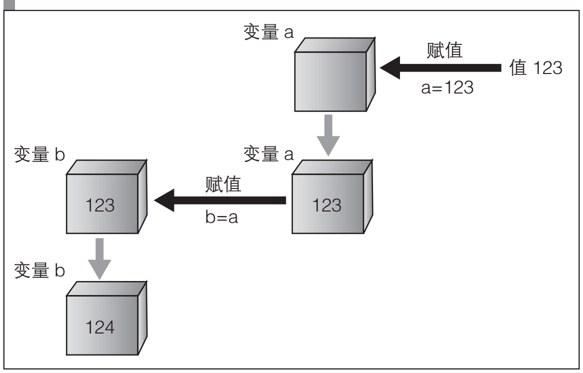
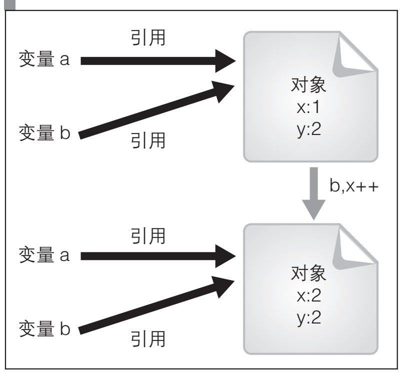


图 5.2　引用类型的变量的执行方式



如果像下面这样，改变了变量 b 所引用的对象，那么这一改变也会体现在变量 a 之中，这是因为这两个变量通过引用而指向了同一个对象。图 5.2 对这种执行方式进行了说明：

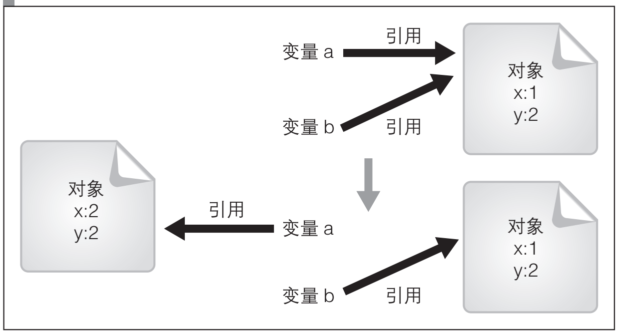
```
// 接上面的代码
js> b.x++;                  // 改变变量 b 所引用的对象
js> print(b.x);             // 变量 b 所引用的对象
2
js> print(a.x);             // 可以发现变量 a 所引用的对象也被改变
2
```

在比较了这两种赋值后，你可能会错误地认为对于值型变量而言，变量值的改变对于其他的变量来

说是不可见的，而对于引用类型的变量，这一改变则是可见的。这是一种不正确的理解。对于引用类型的变量，整个过程中发生改变的其实是其引用的对象，而不是该变量的值。引用类型的变量具有的值就是引用（值），这个值将在赋值的时候被复制。请看下面的代码以及图 5.3。

```
js> var a = { x:1, y:2 };
js> var b = a;             // 变量 a 与变量 b 引用的是同一个对象
js> a = { x:2, y:2 };      // 改变了变量 a 的值（使其引用了另一个对象）
js> print(b.x);            // 变量 b 所引用的对象没有发生改变
1
```

图 5.3　引用类型的变量的执行方式



在 JavaScript 中，赋值运算总是会把右侧的值复制给左侧。对于引用类型的变量来说也是一样，会将引用（用于指示对象的一种值）赋值给左侧。函数调用过程中的参数也是这样的执行方式。在下一节中将对此进行详细说明。

5.2.1　函数的参数（值的传递）

如果你已经理解了上一节的内容，那么对于函数的参数就不需要特别说明了，因为两者的原理是相同的。话虽如此，不少读者在刚接触，遇到将值或者引用作为参数传递给函数的问题时还是会感到迷惑，所以在此仍将对此进行一些说明。

代码清单 5.1 是一个典型的例子，这段代码试图交换两个参数的值，却以失败告终。no_swap 函数的代码试图交换所传递的两个参数 a 与 b 的值。然而，即使调用了这个函数，也不会对实参 one 和 zero 的值造成任何影响。可以认为，在调用函数时执行了相当于 a=one 以及 b=zero 的两次赋值操作。变量 one 与 zero 不会发生变化的理由，已经在前面一节中说明了。就像上一节最后所说的那样，虽然变量 one 与 zero 是引用类型的变量，但实际上也只是对其引用进行了复制操作。因此，并无法实现对 one 和 zero 所引用的对象的交换。

代码清单 5.1　一个无法交换其两参数的值的函数

```
function no_swap(a, b) {
    var tmp = a;
    a = b;
    b = tmp;
}
```

```
// 代码清单 5.1 的运行结果
js> var one = 1;
js> var zero = 0;
js> no_swap(one, zero);
js> print(one, zero);            // 变量 one 与 zero 的值没有发生改变
1 0
```

在前一节的最后说到，在 JavaScript 中，应该把赋值运算看作将右侧的值复制给左侧的一种操作。而这一原则，对于调用函数过程中，参数对引用进行复制的情况也是成立的。这样的规则被称为按值传递（call-by-value）[①]。

在支持对引用或指针进行运算的语言中，可以以代码清单 5.1 中函数的形式，来对实参的值进行交换。JavaScript 不支持这样的功能，所以必须通过其他方式来实现对两个参数值的交换。可以通过传递一个数组并交换其中的元素，或者通过传递一个对象并交换其属性值之类的形式来实现。代码清单 5.2 使用了 JavaScript 自带的增强功能，将交换结果设为函数的返回值，这可以说是一种最为简单的实现代码。

代码清单 5.2　一个能够交换两个参数的值的函数（JavaScript 自带的增强功能）

```
function swap(a, b) {
    return [b, a];
}
```

```
// 代码清单 5.2 的运行结果
js> [one, zero] = swap(one, zero);
js> print(one, zero);
0 1
```

5.2.2　字符串与引用

在此对字符串与引用的关系进行说明。将字符串值赋值给变量时，究竟是复制了字符串的值呢，还是复制了其引用呢？

字符串型是一种基本数据类型，根据语法规则，对其值本身进行复制时对一致性的要求更高。同时，由于比较运算判断的正是字符串的内容是否一致，所以将其认为是一种值也会更加易于理解。然而，在具体实现语言时，几乎所有的 JavaScript 实现都采用了引用复制的方式。这是因为，如果在进行变量赋值时进行字符串值的复制，效率将变得非常低。

那么，是否可以把字符串型看作一种引用类型呢？答案是肯定的，将其看作值的类型也好，引用类型也好，都不会有问题，因为字符串型是一种不可变类型。由于字符串值是无法改变的，因此不管是对其值本身进行复制，还是对其引用进行复制，表面上并不会有什么区别。

总而言之，即使字符串型在内部是以引用类型的方式实现的，从语言规则上来看它仍然是一种值的类型。不过以字符串对象（String 类的对象实例）赋值的变量，从语言规则上来看则是一种引用类型。

5.2.3　对象与引用相关的术语总结

在将对象的引用赋值给变量 a 时，这个对象将被称作“对象 a”。这种称法，会给读者一种（本应不具有名字的）对象其实具有 a 这样一个名称的感觉。显然这样的感觉是不正确的，因为这个对象即使在没有变量 a 的情况下，也能够独立存在。这样说的证据是，如果将变量 a 消去，或是将变量 a 指向其他的对象，原来的这个对象仍然会存在[②]。话虽如此，每次都很准确地使用“变量 a 所引用的对象”这样的说法过于冗长，所以方便起见，还是称其为对象 a。

此外，在上下文不会发生误会的情况下，可以用“对象”这一术语来指代“对象的引用”。对象是一个实体，而引用是用于指示这一实体的位置信息，两者本应是不同的。不过根据上下文可以知道，“将对象赋值给变量 a”的说法很显然是指将对象的引用赋值，所以方便起见可以直接这么说。

① 在 JavaScript 中将引用类型的变量作为参数传递时，实际上传递的是引用。另外还有一种名为按引用传递（call-by-reference）的说法，很容易与此发生混淆，请注意这和引用的按值传递是不同的概念。

② 事实上没有被任何变量引用的对象是会被内存自动回收的，不过这已经是另一个话题了。

5.3 变量与属性

也许很多读者都会觉得对象的属性和变量非常相似吧。两者都可以通过其名字（变量名或属性名）来获取其值，也都可以作为赋值对象，而写在赋值表达式的左侧。其实，在 JavaScript 中变量就是属性，两者何止是相似，本身就是同一个概念。

根据作用域的不同，变量可以被分为全局变量和局部变量（包括参数变量）。全局变量是在最外层代码中声明的变量。所谓最外层代码，指的是写在函数之外的代码。局部变量则是在函数内部声明的变量。全局变量和局部变量两者的本质都是属性。

全局变量（以及全局函数名）是全局对象的属性。全局对象是从程序运行一开始就存在的对象。详细的内容将会在之后的 5.21 节中进行说明。可以通过下面的方式，来实证全局变即为全局对象的属性。

```
js> var x = 'foo';            // 对全局变量 x 进行赋值
js> print(this.x);            // 可以通过 this.x 进行访问
foo

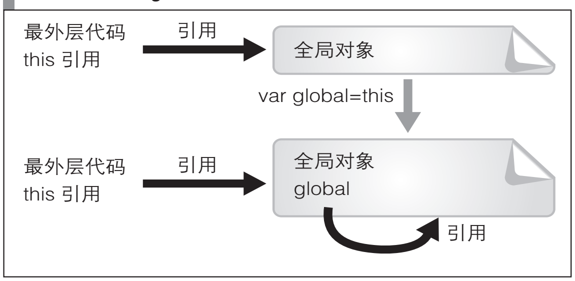
js> function fn() {};         // 全局函数。函数内容在此没有影响，所以留空。
js> 'fn' in this;             // 全局对象的属性 fn
true
```

最外层代码中的 this 引用是对全局对象的引用。因此上面代码中的 this.x，指的就是全局对象的属性 x，这也就是全局变量 x。

像下面这样，在最外层代码中将 this 引用的值赋值给全局变量 global 的话，这个变量就不但是全局对象的属性，同时也是一个对全局对象的引用，从而形成了一种自己引用自己的关系（图 5.4）。

```
js> var global = this;        // 将 this 引用赋值给全局变量 global
js> 'global' in this;         // 全局对象的属性 global
true
```

图 5.4 属性 global 具有一种自己引用了自己的关系



这种关系看起来有些混乱，在 JavaScript 中却很常见。如果是客户端 JavaScript，将会在一开始就提供一个引用了全局对象的全局变量 window。全局对象与变量 window 的关系，和之前例子中的变量 global 是相同的。

在函数内声明的变量是局部变量。作为函数参数的参数变量也是一种局部变量。局部变量（以及参数变量）是在调用函数时被隐式生成的对象的属性。被隐式生成的对象称为 Call 对象。局部变量通常在从函数被调用起至函数执行结束为止的范围内存在。

之所以说是“通常”，是因为有些局部变量在函数执行结束后仍然可以被访问。这将在之后有关闭包的章节中进行说明。

5.4 变量的查找

从代码的角度来看，（作为右值）写出变量名以对该值进行获取的操作，或者写在赋值表达式左侧以作为赋值对象进行查询的操作，都被称为对变量名称的查找。

因此，在最外层代码中对变量名进行查找，就是查找全局对象的属性。这其实只是换了一种说法，在最外层代码中能够使用的变量与函数，只有全局变量与全局函数而已。

至于对函数内的变量名的查找，前一节中已经介绍过，是按照先查找 Call 对象的属性，再查找全局对象的属性来进行的。这相当于在函数内可以同时使用局部变量（以及参数变量）与全局变量。对于嵌套函数的情况，则会由内向外依次查找函数的 Call 对象的属性，并在最后查找全局对象的属性。

这里使用了“查找变量名”这一说法，较为抽象，而能更直观体现其意义的词则是变量的作用域，其具体论述请参见第 6 章。

5.5 对变量是否存在的检验

如果试图读取没有被声明的变量，则会引起 ReferenceError 异常，这是一种错误，必须对代码进行修正。避免 ReferenceError 异常的一种方法就是在 5.1 节中提到的方法：

```
var a = a || 7;   // 一种习惯用法。如果变量 a 已经具有某值，则使用变量 a 的值
```

该代码利用了对已经声明的变量再次声明不会产生副作用的特性。像下面这样，分成两行并使用不同的变量，作用是一样的。

```
// 如果变量 a 已经具有某值，则使用变量 a 的值。代码示例（1）
var a;
var b = a || 7;
```

准确地说，这一代码并没有判断变量 a 是否已经被声明。例如在该例中，如果变量 a 的值是 0 或者是 ''（空字符），它在被转换为布尔型之后值就会为假，这时，代码中的变量 b 则会被赋值为 7。

接下来的代码可能有些冗长，它直接判断变量 a 的值是否是 undefined 值，由此判断出变量 a 是否已声明，或者是否在声明后值为 undefined。

```
// 如果变量 a 已经具有某值，则使用变量 a 的值。代码示例（2）
var a;
var b = a !== undefined ? a : 7;
```

虽说对同一变量再次声明不会有副作用，但每次都要写一遍 var a 也有些麻烦。为了避免这一问题，可以通过 typeof 运算来判断是否为 undefined 值。

请看下面的例子。这个例子利用了在 JavaScript（ECMAScript）中没有块级作用域的特性。

```
// 如果变量 a 已经具有某值，则使用变量 a 的值。代码示例（3）
//（不使用 var a 的版本）
if (typeof a !== 'undefined') {
    var b = a;
} else {
    var b = 7;
}
// 从这里开始可以使用变量 b
```

在以上这些代码中，无法区分变量 a 是还没声明，还是已经声明但值为 undefined。先不论是否有必要对此加以区分，最后再介绍一种能够区分这两种情况的方法。

在读取未声明变量的值时会引起 ReferenceRrror 异常，所以不可以读取这一变量的值，但是可以仅

对这一名称是否存在进行确认。为此需要使用 in 运算。

可以在最外层代码中，像下面这样来判断在全局对象中是否存在属性 a，也就是说，可以用来检测全局变量 a 是否存在。

```
// 用于判断变量 a 是否已经被声明的代码
if ('a' in this) {
    var b = a;
} else {
    var b = 7;
}
// 从这里开始可以使用变量 b
```

对属性是否存在的检验

正如 5.3 节所述，变量与属性实质上是一样的。不过，如果变量或属性本身不存在，处理方式则会有所不同。请看下面的例子：

```
js> print(x);              // 访问未声明的变量会导致 ReferenceError 异常
ReferenceError: x is not defined
js> print(this.x);         // 访问不存在的属性并不会引起错误
undefined

js> var obj = {};
js> print(obj.x);          // 读取不存在的属性仅会返回 undefined 值，并不会引起错误
undefined
```

读取不存在的属性仅会返回 undefined 值，而不会引起错误。但是如果对 undefined 值进行属性访问的话，则会像下面这样产生 TpyeError 异常。

```
js> print(obj.x.y);
TypeError: obj.x is undefined
```

为了避免产生 TypeError 异常，一般会使用下面的方法。

```
obj.x && obj.x.y
```

但如果是为了检测对象内是否存在某一属性，还请使用 in 运算符。

5.6 对象的定义

5.6.1 抽象数据类型与面向对象

如果从形式上来定义 JavaScript 的对象，它就是一种属性的集合。所谓属性，即名称与值的配对。属性值可以被指定为任意类型的值，包括数组或其他的对象，都没有问题。之后还会说到，属性值甚至还可以是一个函数。

面向对象是一种程序设计方法，它已经被广泛接受，如今这已经不再仅仅是一种方法，而成为了一种思想。本书并不会对此深入探讨，仅对作为一种程序设计方法的面向对象技术进行说明。尽管作了这样的限定，面向对象仍具有多种含义。对于对象有一种很常见的定义，即它是一种数据和操作（子程序）的结合。这一定义可以理解为，将面向对象看作一种抽象数据类型的表现形式。

这种理解方式被 C++ 或 Java 等语言所采用，是现在相对主流的见解。想必有很多读者都听说过面向对象的 3 要素，即封装、继承与多态吧。如果这样理解的话，面向对象程序设计的焦点就在于对象的执行方式，并将执行方式的共性定义为一种类型。

在这一语境中，常常使用类这一术语来表达类型的含义。也有些语言会把执行方式与其实现分开，

将执行方式定义为接口。接口的实例（实体）被称为对象，可以对其进行指定的操作。

5.6.2　实例间的协作关系与面向对象

另一种面向对象程序设计的观点认为，与其考虑执行方式之间的共性，更应该关注实例之间的协作关系，即所谓的对象是进行消息收发的实体[①]。对象收到消息之后将会对其作出响应。从实现的角度来看，消息的实质就是通过对方法（函数）进行调用，将对消息的响应分派给方法来处理。从本质上来说，面向对象这一术语只不过是一种在高于内部实现的语境中所使用的、较为抽象的概念而已。打个比方，可以把消息当作一种通信协议，把对象当作一个 Web 应用。

5.6.3　JavaScript 的对象

JavaScript 语言所支持的面向对象与后者的理解更为相近。在 JavaScript 中，一切都是对象。对象之间的协作（消息收发）通过属性访问（以及方法的调用）来实现。而对象之间的共性，则是通过继承同一个对象的性质的方式来实现。JavaScript 通过基于原型的形式来实现继承。

一旦要对面向对象的概念进行说明，事情就会变得很抽象。如果只考虑具体该如何使用 JavaScript 的对象，就不必考虑那么多复杂的问题。只需要考虑最核心的内容，将其理解为在程序中可以进行操作的数据的一种扩充即可。此外，还可以通过函数方法的形式来表示对数据进行操作的子程序。这种想法的核心就是将对象的功能进行拆分并分别进行处理。分割本身也只不过是一种手段。毕竟，面向对象方法的最终目的是降低程序的复杂程度。

5.7　对象的生成

5.7.1　对象字面量

在 JavaScript 程序中，如果要使用对象，就需要首先生成该对象。其中一种方法是通过对象字面量来实现对象的生成。在下一节中会提到，在代码中灵活运用对象字面量，更符合 JavaScript 的编程风格。关于对象字面量的语法结构，请参见 2.5.2 节。

下面列举了一些可以使用对象字面量的情况。请注意这里并没有作严格的分类。

- 作为 singleton 模式的用法。
- 作为多值数据的用法（函数的参数或返回值等）。
- 用于替代构造函数来生成对象。

■ 作为 singleton 模式的用法

在设计模式中有一种 singleton 模式。在基于类的开发过程中，这种模式可以将类的实例数限定为 1 个。

之后也会提到，JavaScript 可以实现基于类的程序设计，不过通常会作如下约定：若只需一个对象实例，则不会去设计一个类，而是会使用对象字面量。对类（构造函数）进行设计以实现 singleton 模式的想法完全是一种基于类的思考方式，在 JavaScript 中我们只需直接使用对象字面量即可。

■ 作为多值数据的用法

可以通过对象字面量来实现多值数据。这种用法与作为关联数组的对象是相通的。例如，在代码清单 5.3 中有一个需要三个参数的函数，对参数是否为数值型的判断已被省略。

① 从历史上来看，这才是最初的面向对象的设计思想。不过我个人并不认为较早出现的观点就是了不起的或是更为正确的。

代码清单 5.3　接受多个参数的函数

```
function getDistance(x, y, z) {
    return Math.sqrt(x * x + y * y + z * z);
}
```

```
// 调用代码清单 5.3 中的函数的示例
js> getDistance(3, 2, 2);
```

这一功能还能像代码清单 5.4 那样，通过对象字面量来实现。同样地，参数的类型检测已被省略。

代码清单 5.4　接受对象的函数

```
function getDistance(pos) {
    return Math.sqrt(pos.x * pos.x + pos.y * pos.y + pos.z * pos.z);
}
```

```
// 调用代码清单 5.4 中的函数的示例
js> getDistance({ x:3, y:2, z:2 });
```

很难说哪一种方法更好，两者各有千秋。参数的数量为 3 个的情况有些微妙，或许认为代码清单 5.3 中的方法更为简单的读者会更多一些。

不过，当参数的数量越来越多时，代码清单 5.4 中的方法的优势就会体现出来。如果用代码清单 5.3 中的方法，参数数量增加之后，弄错实参的排列顺序的可能性也会上升，而 JavaScript 这样的动态程序设计语言对参数类型的检测很弱。如果像代码清单 5.4 这样使用对象作为参数，实参以对象字面量的方式传递，就不需要考虑排列的顺序，只需要使用名称即可。在其他一些程序设计语言中，支持对参数进行命名的功能，这种功能也具有类似的优点。

在 JavaScript 中，有一种模拟出默认参数的效果的习惯用法（代码清单 5.5）。这种方法需要与使用对象作为参数的方式结合使用才能发挥效果。所谓默认参数，是指在调用函数时如果没有实参，或是传递了 null，则会传递一个指定的值。JavaScript 并不支持默认参数这一功能，但可以通过代码清单 5.5 这样的形式来实现。

通过 || 运算可以将参数作为布尔型来判断真假，其中利用了若调用函数时没有实参参数的值则为 undefined 这一特性。通常来说，在函数内对参数进行赋值不是一种好习惯（不仅是 JavaScript，所有的程序语言都是如此），不过下面的做法被当作了一种习惯用法。

代码清单 5.5　模拟了默认参数的效果的习惯用法

```
function getDistance(pos) {
    pos = pos || { x:0, y:0, z:0 };// 如果没有收到参数 pos 的话，则使用默认值
    return Math.sqrt(pos.x * pos.x + pos.y * pos.y + pos.z * pos.z);
}
```

不但可以通过对象字面量来方便地实现函数参数的多值数据传递，还可以通过对象字面量，方便地实现函数的多值数据返回。在实际编程中这很常用。虽然直接在代码中书写数值没有什么意义，不过从形式上来说就是代码清单 5.6 这样的，所以，只要将它看作经过了实际的函数处理后，所得到的需要被返回的结果即可。

代码清单 5.6　返回多值数据的函数

```
function fn() {
    省略
    return { x:3, y:2, z:2 };
}
```

```
// 调用代码清单 5.6 中的函数的示例
js> var pos = fn();
js> print(pos.x, pos.y, pos.z);
```

```
3 2 2
```

■ 用于代替构造函数的用法

最后我们介绍一下通过对象字面量来实现一个用于替代构造函数的函数的用法。该函数的功能是生成一个对象，所以需要以对象字面量作为返回值，从形式上来说，它和返回多值数据的函数是相同的。根据狭义的面向对象的定义，多值数据与对象的区别仅在于是否具有特定的执行方式。

和代码清单 5.6 一样，直接在代码中书写数值没有什么意义，这里仅仅是作为一个例子用于说明而已（代码清单 5.7）。

代码清单 5.7　用于生成对象的函数（还有改进的余地）

```
function createObject() {
    return { x:3, y:2, z:2,
         getDistance:function() {
             return Math.sqrt(this.x * this.x + this.y * this.y + this.z * this.z);
         }
    };
}
```

```
// 调用代码清单 5.7 中的函数的示例
js> var obj = createObject();
js> print(obj.getDistance());
4.123105625617661
```

下一节将会介绍 JavaScript 通过构造函数来生成对象的功能。使用返回对象字面量的函数，与通过 new 表达式来调用构造函数，是两种不同风格的生成对象的手段。不过，通过代码清单 5.7 的方法生成的对象，和 5.7.2 节中通过代码清单 5.9 的方法生成的对象，有同样的不足。具体的改善方法将在之后详述。

专栏

JavaScript 中用于函数返回多个值的增强功能

通过 JavaScript 1.7 的增强功能，可以像下面这样，通过数组实现将返回值逐个返回的功能。

```
js> function f() {
        return [3,4,5];
    }

js> var x, y, z;
js> [x,y,z] = f();
js> print(x, y, z);
3 4 5
```

5.7.2　构造函数与 new 表达式

构造函数是用于生成对象的函数。之后会再详述函数与构造函数的区别，这里首先介绍一个具体例子（代码清单 5.8）。

可以直观地将代码清单 5.8 理解为 MyClass 类的类定义。在调用时通过 new 来生成一个对象实例。

代码清单 5.8　构造函数的例子

```
// 构造函数(类的定义)
function MyClass(x, y) {
    this.x = x;
    this.y = y;
}
```

```
// 对代码清单 5.8 的构造函数的调用
js> var obj = new MyClass(3, 2);
js> print(obj.x, obj.y);
3 2
```

从形式上来看，构造函数的调用方式如下。

- 构造函数本身和普通的函数声明形式相同。
- 构造函数通过 new 表达式来调用。
- 调用构造函数的 new 表达式的值是（被新生成的）对象的引用。
- 通过 new 表达式调用的构造函数内的 this 引用引用了（被新生成的）对象。

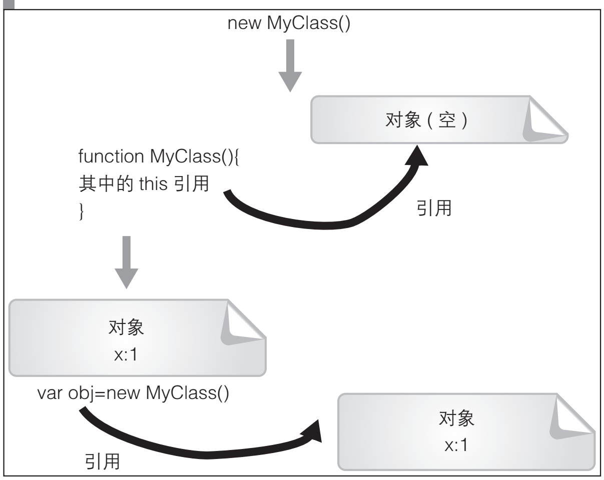
■ new 表达式的操作

在此说明一下 new 表达式在求值时的操作。首先生成一个不具有特别的操作对象。之后通过 new 表达式调用指定的函数（即构造函数）。构造函数内的 this 引用引用了新生成的对象。执行完构造函数后，它将返回对象的引用作为 new 表达式的值。

new 表达式的操作就是以上这些。实际上其中还含有一个和原型链有关的问题，将会在之后进行说明。

图 5.5 说明了构造函数的内部操作。

图 5.5 构造函数的操作图



■ 构造函数调用

构造函数总是由 new 表达式调用。为了与通常的函数调用相区别，将使用 new 表达式的调用，称为构造函数调用。构造函数与通常的函数的区别在于调用方式不同。任何函数都可以通过 new 表达式调用，因此，所有的函数都可以作为构造函数。也就是说，如果一个函数通过函数调用的方式使用，则是一个函数；如果通过构造函数调用的方式使用，则是一个构造函数。在实际开发中，通常会分别设计用于函数调用的函数与用于构造函数调用的函数，所以方便起见，将为了构造函数调用而设计的函数称为构造函数。构造函数的名称一般以大写字母开始（如 MyClass）。

构造函数在最后会隐式地执行 return this 操作。那么，如果在构造函数中显式地写有 return 语句，会发生什么情况呢？结果可能不容易理解。通过 return 返回一个对象之后，它将成为调用构造函数的 new 表达式的值。也就是说，使用 new 表达式后返回的，可能是所生成的对象以外的其他对象。然而，如果调用的构造函数中的 return 返回的是基本类型的值，则会无视这一返回值，仍然隐式地执行 return this 操作。

这种操作常常会造成混乱，我们建议不要再在构造函数内使用 return 语句。

5.7.3　构造函数与类的定义

经过上一节的说明，熟悉 Java 或 C++ 等支持类定义的语言的开发者，可能会觉得 JavaScript 的构造函数有些奇特。毕竟通过 new 表达式调用普通的函数并生成一个对象，是一种不容易理解的语言特性。不过，这已经满足了类定义所必需的功能。

代码清单 5.9 是一个实现了定义一个具有域与方法的类的构造函数的例子。

代码清单 5.9　模拟类定义（尚有改进的余地）

```
// 相当于类的定义
function MyClass(x, y) {
    // 相当于域
    this.x = x;
    this.y = y;
    // 相当于方法
    this.show = function() {
        print(this.x, this.y);
    }
}
```

```
// 对代码清单 5.9 中的构造函数的调用（实例生成）
js> var obj = new MyClass(3, 2);
js> obj.show();
3 2
```

只要按照代码清单 5.9，就能够从形式上实现 JavaScript 的类定义。不过，代码清单 5.9 作为类的定义还存在以下两个问题。前者可以通过原型继承来解决，而后者可以通过闭包来解决。之后会再分别详述。

- 由于所有的实例都是复制了同一个方法所定义的实体，所以效率（内存效率与执行效率）低下。
- 无法对属性值进行访问控制（private 或 public 等）。

5.8　属性的访问

生成的对象可以通过属性来访问。对于对象的引用可以使用点运算符（.）或中括号运算符（[]）来访问其属性。需要注意的是，在点运算符之后书写的属性名会被认为是标识符，而中括号运算符内的则是被转为字符串值的式子。请看下面的例子：

```
js> var obj = { x:3, y:4 };
js> print(obj.x);          // 属性 x
3
js> print(obj['x']);       // 属性 x
3
js> var key = 'x';
js> print(obj[key]);       // 属性 x（而非属性 key）
3
```

不过，对于对象字面量的属性名来说，下面这样的标识符或字符字面量形式的表示，都没问题。请注意不要与上面的规则混淆。

```
js> var key = 'x';
js> var obj = { key:3 };   // 属性 key（而非属性 x）
js> var obj = { 'x':3 };   // 属性 x
```

这里需要多提一句，属性访问的运算对象并不是变量，而是对象的引用。这一点，可以从以下直接对对象字面量进行运算的示例中得到确认：

```
js> ({x:3, y:4}).x;        // 属性 x
3
```

```
js> ({x:3, y:4})['x'];     // 属性 x
3
```

现实中几乎不会对对象字面量进行运算。不过当这种运算对象不是一个变量时，倒是常常会以方法链之类的形式出现。

5.8.1 属性值的更新

在赋值表达式的左侧书写属性访问表达式能够实现对属性值的改写。如果指定的是不存在的属性名，则会新增该属性。下面将不再使用右侧或左侧的说法，而改用属性读取，以及属性写入这样的术语。

可以使用 delete 运算表达式来删除属性。这里需要注意的是，很难区分不存在的属性与属性值为 undefined 值的属性。关于 delete 运算以及判断属性是否存在的方法，请参见 5.9 节。

5.8.2 点运算符与中括号运算符在使用上的区别

有时选择用于访问对象属性的这两个运算符只凭偏好。点运算符的表述较为简洁，所以通常都会选用点运算符。不过，中括号运算符的通用性更高。

能使用点运算符的情况一定也可以使用中括号运算符，反之未必成立。但也无需因此全都使用中括号运算符。通常默认使用表述简洁的点运算符，只有在不得不使用中括号运算符的情况下，才使用中括号运算符。

只能使用中括号运算符的情况分为以下几种。

- **使用了不能作为标识符的属性名的情况。**
- **将变量的值作为属性名使用的情况。**
- **将表达式的求值结果作为属性名使用的情况。**

包含数值或横杠（-）的字符串不能作为标识符使用。无法作为标识符使用的字符串，不能用于点运算符的属性名，且对于保留字，也有这样的限制。不过，原本就不应该将保留字作为属性名使用，所以这里不再赘述。

像下面这样，将含有横杠的属性名用于点运算符会引起错误。

```
// 含有横杠的属性名
js> obj = { 'foo-bar':5 };

js> obj.foo-bar;      // 将解释为 obj.foo 减去 bar，从而造成错误
ReferenceError: bar is not defined
```

无法作为标识符被使用的字符串，仍可以在中括号运算符中使用。请看下面的例子，其中以字符串值指定了一个属性名。

```
js> obj['foo-bar'];   // 使用 [] 运算以字符串值指定了一个属性名。可以正常执行
5
```

数值也是如此。数组对象的属性名都是数值。由于点运算符无法使用数值，因此只能使用中括号运算符。而且很多程序设计语言都是通过中括号运算符来访问数组的元素，所以可读性也随之提高。

下面的例子仍使用了之前的代码，用于展示将被变量的值作为属性名使用的情况。

```
js> var key = 'x';
js> obj[key];         // 属性 x（而非属性 key）
```

如果表达式的求值结果是字符串，可以直接用中括号运算符通过该表达式指定属性名。下面引用出自《JavaScript 语言精粹》[①] 一书的一个具有一定技巧性的例子。

---

① *JavaScript: The Good Parts*，作者 Douglas Crockford，中文版由电子工业出版社出版，赵泽欣、鄢学鹍译。——译者注

这段代码会根据数值的符号而选择调用不同的方法。方法调用一词会让人觉得要使用的是点运算符，不过事实上中括号运算符也能被调用。

```
// 引用自《JavaScript 语言精粹》一书
// 仅读取数值的整数部分的处理
Math[this < 0 ? 'ceiling' : 'floor'](this)
```

5.8.3　属性的枚举

可以通过 for in 语句对属性名进行枚举（代码清单 5.10）。通过在 for in 语句中使用中括号运算符，可以间接地实现对属性值的枚举。使用 for each in 语句可以直接枚举属性值。

代码清单　5.10 属性的枚举

```
var obj = { x:3, y:4, z:5 };
for (var key in obj) {
    print('key = ', key);          // 属性名的枚举
    print('val = ', obj[key]);     // 属性值的枚举
}
```

```
// 代码清单 5.10 的运行结果
key = x
val = 3
key = y
val = 4
key = z
val = 5
```

属性可以分为直接属性以及继承于原型的属性。for in 语句和 for each in 语句都会枚举继承于原型的属性。其中的区别以及 for in/ for each in 语句以外的属性枚举方法，将在之后的 5.17.5 节中说明。

5.9　作为关联数组的对象

在 2.5 节提到过，JavaScript 的对象和 Java 的映射（Map）类似。5.7.1 节也介绍了作为多值数据的对象。

如果将 JavaScript 对象的属性名看作键，属性值看作值，我们会发现它与 Java 中的映射非常相似。JavaScript 的对象还具有 Java 的映射所不具备的附加功能（例如方法或原型继承等），但也可以不理会这些功能，直接将其作为映射来使用。

5.9.1　关联数组

首先对与关联数组相关的术语进行整理。将数值作为键的值的数据结构通常称为数组。数组是绝大多数程序设计语言都支持的一种基本的数据结构[①]。

由于数组的键是连续的数值，因此可以将其看作具有顺序的值的集合。除了数值以外大多都会使用字符串作为键值。不过键的类型也可以不限于字符串，对任意类型的键与值的集合进行操作的数据结构称为关联数组。在有些语言中，关联数组也被称为映射或字典。也有根据内部实现而将其称为散列的语言。虽然用词不同，但其数据结构是相同的，使用何种称法都可以。本书将使用关联数组这一术语。

关联数组最主要的用途是执行通过键来读取值的操作。在其他程序设计语言，特别是一些脚本语言中，关联数组被设计为一种语言本身的功能，不过在 JavaScript 中，必须通过对象来实现关联数组。

① JavaScript 中的数组将在之后的 7.1 节中进行说明。

本节将阐述作为关联数组的对象。请注意，并没有专门用于关联数组的对象，这仅仅是对对象的一种不同的用法。

■ 关联数组的操作方式

关联数组是元素的集合，其元素为键与值的配对。关联数组的基本操作有通过键来获取值、元素的设定、元素的删除这 3 种。由于其实体是 JavaScript 的对象，所以这里的元素只不过是属性的另一种说法，而键与值分别是属性名与属性值的别称。

可以按照属性访问一节中的介绍，通过点运算符或中括号运算符来实现按键取值。严格地说，是将该值作为右值来使用。

对于元素的设定，可以将点运算符或是中括号运算符作为左值写入赋值表达式。具体的例子请参见 5.7.1 节。

对象的删除可以通过 delete 运算符。用对象的术语来说就是删除属性。使用方法如下。

```
// 删除关联数组的元素的例子（属性的删除）

js> var map = { x:3, y:4 };
js> print(map.x);
3
js> delete map.x;      // 也可以使用 delete map['x']
true                   // 如果删除成功，则返回 true
js> print(map.x);      // 如果读取已被删除的元素，则返回 undefined 值
undefined
```

在 C++ 语言中也有 delete 这个关键字，不过其功能却全然不同。在 C++ 中 delete 的功能是释放所引用的对象的内存，而在 JavaScript 中 delete 只用于删除对象中的属性。用映射中的术语来说就是，仅仅从映射中删除键，使其对应的值（对于对象来说也就是属性值）与该键不再有对应关系。虽然失去了引用的对象最终可能会因为垃圾回收机制而消失，不过这并不是 delete 运算的直接功能。

对不存在的元素进行访问得到的结果是 undefined 型。需要注意的是，这与 Java 中映射返回的 null 是不同的。由于可以显式地将值设定为 undefined 值，因此无法通过将键与 undefined 值作等值比较来实现对键是否存在的检验。关于对键是否存在的检验，将在之后的 5.9.2 节说明。

可以通过 for in 语句对键进行枚举。详细的说明请参见 5.8.3 节。

5.9.2 作为关联数组的对象的注意点

作为关联数组的对象有一些和原型继承相关的注意点。原型继承的概念将在之后详述，简单说来，原型继承指的是一种对象继承其他对象的属性并将其作为自身的属性一样来使用的做法。

如下所示，从形式上来说，对象 obj 的属性并不是其直接属性，而是通过原型继承而得到的属性。

```
js> function MyClass() {}
js> MyClass.prototype.z = 5;           // 在原型链上设定属性 z

js> var obj = new MyClass();           // 属性 z 继承了原形
js> print(obj.z);
5
```

for in 语句将枚举通过原型继承而得到的属性。

```
// 接之前的代码
js> for (var key in obj) { print(key); }   // for in 语句也会枚举通过原型继承得到的属性
z
```

请注意，通过原型继承而得到的属性无法被 delete。继续接之前的代码。

```
// 接之前的代码
js> delete obj.z;                      // 尽管没有被 delete，但还是会返回 true……
```

```
true
js> print(obj.z);                    // 无法 delete 通过原型继承而得到的属性
5
```

在将对象作为关联数组使用时，通常都会使用对象字面量来生成。不过需要注意的是，即使视图通过使用空的对象字面量以创建一个没有元素的关联数组，也仍然会从 Object 类中继承原型的属性。可以通过 in 运算对此进行检验。

```
js> var map = {};                    // 通过空的对象字面量生成关联数组
js> 'toString' in map;               // 从 Object 类中原型继承了属性 toString
true
```

但是，通过 for in 语句对元素进行枚举不会有任何效果。这是由于 enumerable 属性的缘故，将在之后的小节中说明。请参见 5.17.5 节。

```
// 接之前的代码
js> for (var key in map) {
        print(key);
    }
// 没有元素会被枚举
```

通过 in 运算符检测关联数组的键是否存在，就会发生与原型继承而来的属性相关的问题。因此，像下面这样通过 hasOwnProperty 来对其进行检测，是一种更安全的做法。

```
js> var map = {};
js> map.hasOwnProperty('toString');     // 由于 toString 不是直接属性，因此结果为 false
false

js> map['toString'] = 1;
js> map.hasOwnProperty('toString');
true

js> delete map['toString'];
js> map.hasOwnProperty('toString');
false
```

5.10 属性的属性

虽然说起来有些绕口，不过属性也是有其属性的。表 5.1 总结了 ECMAScript 第 5 版中定义了的属性①。

在 ECMAScript 中，属性值被定位为“值属性”这样一种属性。使用这一定义的话，属性就成为了名称（属性名）和多个属性的集合。本书更侧重于使理解更为直观易懂，所以将分别考虑值与属性的问题。

表 5.1　属性的属性

属性的属性名	含义
writable	可以改写属性值
enumerable	可以通过 for in 语句枚举
configurable	可以改变属性的属性。可以删除属性。
get	可以指定属性值的 getter 函数
set	可以指定属性值的 setter 函数

在表 5.1 的属性中，enumerable 是在 ECAMScript 第 5 版之前就被广泛使用的属性。在标准的对象中有一部分属性的 enumerable 属性为假而无法通过 for in 语句枚举。其中一个很容易理解的例子是数列的 length 属性。

虽然 ECMAScript 第 5 版对属性读写方法进行了标准化处理（参见 5.18 节），不过在实际的

① 对于可能产生歧义的部分，将译为“属性的属性”以示区别。——译者注

JavaScript 开发中，我们一般也不会用到对属性的读写。而 enumerable 也是标准对象所具有的属性，所以通常也不需要对自己生成的对象的属性显式地进行修改。不过属性本身确实有助于使代码更为健壮，或许随着 ECMAScript 第 5 版的普及，变更属性的情况也会变得越来越常见。

5.11 垃圾回收

不再使用的对象的内存将会自动回收，这种功能称作垃圾回收。所谓不再使用的对象，指的是没有被任何一个属性（变量）引用的对象。

由于 JavaScript 有着客户端程序大多运行时间很短这一历史原因，因此与其他程序设计语言相比，开发者并不太关心对象的存在生命周期。如果整个程序的生命周期就很短，相对来说就没有必要对每个对象的生命周期太过在意。

不过随着最近各种 Web 应用以及服务器端 JavaScript 程序的发展，情况发生了变化。现在已经有必要像其他的程序设计语言那样，考虑对象的生命周期问题了。垃圾回收的目的是，使开发者不必为对象的生命周期管理花费太多精力。因此通常只考虑代码即可，具体的 JavaScript 实现会帮忙解决那些麻烦的问题。虽说通过 delete 来删除不再使用的属性是一个不错的习惯，但只要不会造成内存泄漏，就没有必要在这方面花太多的心思。

不过，即使有垃圾回收功能，仍然有可能发生内存泄漏。有些是由于垃圾回收机制的实现存在问题，更多的是因为发生了循环引用的情况而造成了内存泄漏。

所谓循环引用，指的是对象通过属性相互引用而导致它们不会被判定为不再使用的状态。对于客户端 JavaScript 来说，存在几种常见的可能导致循环引用的情况，因此建议使用内存泄漏检测工具来检测。

5.12 不可变对象

5.12.1 不可变对象的定义

所谓不可变对象，指的是在被生成之后状态不能再被改变的对象。由于对象的状态是由其各个属性的值所决定的，因此从形式上来说也是指无法改变属性的值的对象。也有观点认为，在对象引用了另一个对象的情况下，只有当那个被引用的对象也是不可变的时候，引用了它的对象才能被称为不可变对象。

从广义上来说，不可变对象指的是不去改变状态的对象。而从狭义上来说，只有既没有改变，也无法改变状态的对象，即为了禁止改变而专门设计的对象，才被称为不可变对象。JavaScript 中的一种典型的不可变对象就是字符串对象。

5.12.2 不可变对象的作用

灵活运用不可变对象有助于提高程序的健壮性。这是因为，程序中的很多错误都是由于非法改变了对象的状态而造成的。例如，将对象传递给方法的参数时，存在方法会改写对象内容的隐患。如果那是一个不可变对象，则不用担心这一问题。不清楚对象的内部构造就改写很容易引起错误，在排除了这种情况之后，就可以减少花在这个问题上的精力。

虽然不可变对象是一种便利的程序设计技巧，但其实在 JavaScript 开发中并没有被大量使用。其中最主要的一个原因就是花销的取舍。为了确保对象的不可变，不得不增加一些和主要功能无关的代码。对于一直使用小规模代码的 JavaScript 来说，需要权衡花销。本节的最后会再总结一下这个话题。

5.12.3　实现不可变对象的方式

在 JavaScript 中可以通过以下方式实现对象的不可变。

- 将属性（状态）隐藏，不提供变更操作。
- 灵活运用 ECMAScript 第 5 版中提供的函数。
- 灵活运用 writable 属性、configurable 属性以及 setter 和 getter。

JavaScript 中的对象没有像 private 属性这样的显式访问控制功能。为了将属性隐藏，可以使用一种被称为闭包的方法。具体内容将在之后的 6.7.5 节介绍。

在 ECMAScript 第 5 版中有一些用于支持对象的不可变化的函数（表 5.2）。seal 可以向下兼容 preventExtensions，freeze 可以向下兼容 seal。这里的向下兼容，指的是比后者有更为严格的限制。

表 5.2　ECMAScript 第 5 版中用于支持对象的不可变化的函数

方法名	属性新增	属性删除	属性值变更	确认方法
preventExtensions	X	O	O	Object.isExtensible
seal	X	X	O	Object.isSealed
freeze	X	X	X	Object.isFrozen

图 5.6 ～图 5.8 是各个方法的具体示例。Object.keys 方法用于对属性枚举。关于该方法的详细信息，请参见 5.17.5 节。

图 5.6　Object.preventExtensions 的例子

```
js> var obj = { x:2, y:3 };
js> Object.preventExtensions(obj);

// 无法新增属性
js> obj.z = 4;
js> Object.keys(obj);
["x", "y"]

// 可以删除属性
js> delete obj.y;
js> Object.keys(obj);
["x"]

// 可以更改属性值
js> obj.x = 20;
js> print(obj.x);
20
```

图 5.7　Object.seal 的例子

```
js> var obj = { x:2, y:3 };
js> Object.seal(obj);

// 无法删除属性
js> delete obj.y;          // 将返回 false
js> Object.keys(obj);
["x", "y"]

// 可以更改属性值
js> obj.x = 20;
js> print(obj.x);
20
```

图 5.8 Object.freeze 的例子

```
js> var obj = { x:2, y:3 };
js> Object.freeze(obj);

// 无法更改属性值
js> obj.x = 20;
js> print(obj.x);
2
```

对于表 5.2 中的方法，有以下几点需要注意。

- **一旦更改就无法还原。**
- **如果想让原型继承中的被继承方也不可变化，需要对其进行显式的操作。**

从内部实现来看，seal 的作用是将属性的 configurable 属性置为假，而 freeze 是将 writable 属性置为假。关于属性，请参见 5.10 节。如果在生成对象时，对这些属性进行显式地设置，也能够取得相同的效果。具体方法请参见 5.18 节。

灵活运用属性的属性，还能够实现只有 getter 方法而没有 setter 方法的不可变对象[①]。

在本节的最后，我们建议尽可能不使用不可变对象。这个建议听起来太过随意，没有什么帮助，不过确实应该为程序的健壮性与其开销选择一个折中方案。为了安全性而增加开销，产品可能就会无法按时完成。此外，客户端 JavaScript 对代码的体积有着严格的要求，因此过分注重安全性的代码可能反而会降低用户体验。这不是一个简单的是非问题，而是一个需要做出判断的问题。尽管不可变对象是提升代码健壮性的一个有效方法，但如果过分拘泥于此而降低了用户的使用体验，反而本末倒置了。实际的程序开发与理论研究有所不同，请时刻谨记考虑健壮性与开销之间的平衡。

5.13 方法

在 JavaScript 的语言规范中并不存在方法这一概念。方便起见，我们将作为对象属性的函数称为方法。而在实际中可以像这样定义方法：那些使用了 this 引用来调用并访问了对象的属性的函数，被称为方法。方法与函数名称两者可以随意混用，不过如果能注意到正在使用的是一个方法的话，就能更明确地意识到现在是在对对象进行操作。所以，使用方法这一名称的话会更有意义。

5.14 this 引用

this 引用是一种在 JavaScript 的代码中随时都可以使用的只读变量。在 Java 或 C++ 中也有功能类似的 this 引用。在 Java 以及 C++ 中，this 应该被看作是隐式传递的参数，而在 JavaScript 中，this 引用可以在最外层代码（函数之外）使用，所以从直觉上更像是一个可以随时使用的只读变量。

this 引用引用的是一个对象。对于最外层代码与函数内部的情况，其引用目标是不同的。此外，即使在函数内部，根据函数调用方式的不同，引用对象也会有所不同。需要注意的是，this 引用有着会根据代码的上下文语境自动改变其引用对象的特性。

5.14.1 this 引用的规则

在此，总结一下 this 引用的规则。

① 具体的例子在 5.18 节第 5 版中的 Object 类中再为介绍。

- **在最外层代码中，this 引用引用的是全局对象。**
- **在函数内，this 引用根据函数调用方式的不同而有所不同（参见表 5.3）。**

需要注意的是，对于函数内部的情况，this 引用的引用对象并不是根据函数的内容或声明方式而改变的，而是根据其调用方式而改变。也就是说，即使是同一个函数，如果调用方式不同，this 引用的引用对象也会有所不同。

表 5.3　函数内部的 this 引用

函数的调用方式	this 引用的引用对象
构造函数调用	所生成的对象
方法调用	接收方对象
apply 或是 call 调用	由 apply 或 call 的参数指定的对象
其他方式的调用	全局对象

对于构造函数调用的情况，this 引用的引用对象是所生成的对象。详细的内容请参见 5.7.2 节。

表 5.2 的方法调用的说明中的接收方对象是这样一种对象。

- **通过点运算符或中括号运算符调用对象的方法时，在运算符左侧所指定的对象。**

在之前的小节中也提到过，方法是对象的属性所引用的函数。下面是一个关于方法和接收方对象的具体例子。

```
// 对象定义
js> var obj = {
    x:3,
    doit: function() { print('method is called.' + this.x ); }
    };

js> obj.doit();            // 对象 obj 是接收方对象。doit 是方法。
method is called. 3

js> obj['doit']();         // 对象 obj 是接收方对象。doit 是方法。
method is called. 3
```

现在说明上面的例子。首先是将对象的引用赋值给了变量 obj。这个对象有两个属性。属性 x 的值为数值 3，属性 doit 的值是一个函数。将该函数称为方法 doit。

可以通过点运算符或中括号运算符对 obj 调用方法 doit。这时，方法调用的目标对象被称为接收方对象（也就是说，obj 所引用的对象是一个接收方对象）。被调用的方法内的 this 引用引用了该接收方对象。

由于 this 引用具有这样的特性，JavaScript 的方法拥有了与 Java 或 C++ 中的方法相似的功能。之后的小节将阐述它们之间的细微区别。apply 与 call 的作用是用于显式地指定接收方对象，详细内容将在之后的 5.15 节说明。

5.14.2　this 引用的注意点

在之前的小节中说过，虽然 JavaScript 的 this 引用在方法调用中的执行方式和 Java 或 C++ 中的基本相同，但仍应注意它们之间存在一些细微的差别。在 Java 这样的基于类的语言中，this 所引用的接收方对象始终是该类的实例，而在 JavaScript 中，却不一定总是如此。

JavaScript 的 this 引用的引用对象，会随着方法调用方式的不同而改变。下面是一个简单的例子，用于说明通过其他的接收方对象调用某个函数，或是在没有接收方对象的情况下，this 引用的操作是不同的。

```
js> var obj = {
    x:3,
    doit: function() { print('method is called.' + this.x ); }
    };
```

```
js> var fn = obj.doit;            // 将 ojb.doit 引用的 Function 对象赋值给全局变量

js> fn();                          // 函数内的 this 引用引用了全局对象
method is called. undefined

js> var x = 5;                     // 确认 this 引用确实引用了全局对象
js> fn();
method is called. 5

js> var obj2 = { x:4, doit2:fn };  // 将 obj 的方法（Function 对象的引用）赋值给了另一个对象 obj2 的属性
js> obj2.doit2();                  // 方法内的 this 引用引用了对象 obj2
method is called. 4
```

在 Java 中常常可以省略 this，不用明确地写出。因为在查找方法内的名称时总是会在同一个类的域名与方法名中搜索。而在 JavaScript 中就不能像这样省略 this 了。如果在上面的例子中将 this.x 改写为 x，它的含义将会变为全局变量 x。

■ 在方法内部调用方法的情况

在方法内部调用方法时也需要对 this 引用多加注意，下面是一个例子。通常在 Java 或 C++ 中的方法内进行方法调用时会省略 this，然而在 JavaScript 的方法内调用其他的方法时，必须像下面的例子那样通过 this 引用来实现。

```
// 从 doit 方法内调用 doit2 方法时，必须通过 this 引用，以 this.doit2() 的方式实现
js> var obj = {
    x:3,
    doit: function() { print('doit is called.' + this.x ); this.doit2(); },
    doit2: function() { print('doit2 is called.' + this.x); }
    };

js> obj.doit();
doit is called. 3
doit2 is called. 3
```

在上面的例子中，如果将 this.doit2() 写成 doit2()，则会在全局函数中搜索 doit2。在没有语法错误时，嵌套的函数将按作用域由内至外的顺序来查找名称（请参见 5.4 节）。

5.15 apply 与 call

在 Function 对象中包含 apply 与 call 这两种方法，通过它们调用的函数的 this 引用，可以指向任意特定的对象。也就是说，可以理解为它们能够显式地指定接收方对象。

下面是一个使用了 apply 方法与 call 方法的例子。

```
js> function f() { print(this.x); }
js> var obj = { x:4 };

js> f.apply(obj);              // 通过 apply 调用函数 f。函数内的 this 引用引用了对象 obj
4
js> f.call(obj);               // 通过 call 调用函数 f。函数内的 this 引用引用了对象 obj
4

// 将接收方对象指定为另一个对象并进行方法调用
js> var obj = {
    x:3,
    doit: function() { print('method is called.' + this.x ); }
    };

js> var obj2 = { x:4 };

js> obj.doit.apply(obj2);      // 通过 apply 调用 obj.doit 方法。方法内的 this 引用引用了对象 obj2
```

```
method is called. 4
```

对 Function 对象 f 使用 apply 或 call 方法，就能够调用该函数。不考虑函数内的 this 引用的话，这和 f() 的用法是一样的。两者的区别在于被调用的函数（方法）内的 this 引用，this 引用的是作为 apply/call 的第一个参数被传递的对象。而 apply 与 call 之间的不同之处在于两者对其他参数的传递方式。对于 apply 来说，剩余的参数将通过数组来传递，而 call 是直接按原样传递形参。请通过下面具体的例子来了解这一差异。

```
js> function f(a, b) { print('this.x = ' + this.x + ', a = ' + a + ', b = ' + b); }

js> f.apply({x:4}, [1, 2]);    // 作为第 2 个参数的数列中的元素都是函数 f 的参数
this.x = 4, a = 1, b = 2

js> f.call({x:4}, 1, 2);       // 从第 2 个参数起的参数都是函数 f 的参数
this.x = 4, a =1 , b = 2
```

在实际的编程过程中，我们常常会为了函数回调而使用 apply 或 call 调用。详细内容请参见 6.8 节。

5.16　原型继承

本节将阐述原型继承。事实上，原型继承的内部执行方式是相当复杂的。如果只是希望能够使用原型继承，而没有弄清其用法的话，反而可能会导致混乱的局面。因此，首先仅说明一下其形式。按代码清单 5.9 中的类定义为模板，并以原型继承的方式改写，就能得到代码清单 5.11。

代码清单 5.11　使用了原型继承的类定义

```
// 相当于进行类定义
function MyClass(x, y) {
    this.x = x;
    this.y = y;
    }
MyClass.prototype.show = function() {
                   print(this.x, this.y);
                   }
```

```
// 代码清单 5.11 的构造函数调用（实例生成）
js> var obj = new MyClass(3, 2);
// 方法调用
js> obj.show();
3 2
```

代码清单 5.9 与代码清单 5.11 的区别在于，前者的方法定义直接是对象实例的属性，而后者不是。在代码清单 5.11 中，方法 show 并不是对象 obj 的直接属性，但也可以被调用。从表面上来看，它是从另一个对象（MyClass.prototype）的属性继承而来的。这就是对原型继承的一种形式上的理解。

在 JavaScript 中，保存有值的属性和保存有函数的属性之间并没有什么特别的区别，所以除了方法之外其他的值也能够被原型继承。不过在实际中需要进行原型继承的大多是方法。此外，将构造函数名与类名进行替换不会造成什么问题，所以形式上像下面这样使用原型继承即可。

```
// 对原型继承的形式上的理解
类名 .prototype. 方法名 = function( 方法的参数 ) { 方法体 }
```

5.16.1　原型链

原型继承支持一种称为原型链的功能。使用原型链有两个前提。

- 所有的函数（对象）都具有名为 prototype 的属性（prototype 属性所引用的对象则称为 prototype 对象）。

- 所有的对象都含有一个（隐藏的）链接，用以指向在对象生成过程中所使用的构造函数（Function 对象）的 prototype 对象。

在 ECMAScript 的标准中，prototype 属性被称为 explicit prototype property，而隐藏的链接被称为 implicit prototype link。本书将前者称为“prototype 引用”，而将后者称为“隐式链接”。

在满足了以上前提的情况下，原型链将以以下方式运行。

对象对属性的读取（以及对方法的调用）是按照以下顺序查找的。

① 对象自身的属性。

② 隐式链接所引用的对象（即构造函数的 prototype 对象）的属性。

③ 第 2 项中的对象的隐式链接所引用的对象的属性。

④ 反复按第 3 项的规则查找直至全部查找完毕（查找的终点是 Object.prototype 对象）。

如果不考虑原型链这一术语的话，会发现其本质其实就是对隐式链接的属性继承。由于隐式链接所引用的对象是构造函数的 prototype 对象，因此事实上这就是在前面小节中所说的“类名.prototype. 方法名”的继承方式。此外需要注意，由对象字面量生成的对象的隐式链接引用的是 Object.prototype。

而在写入对象的属性的时候，则是按照以下顺序进行属性查找的。这时的属性改写中不会发生继承。

① 对象自身的属性

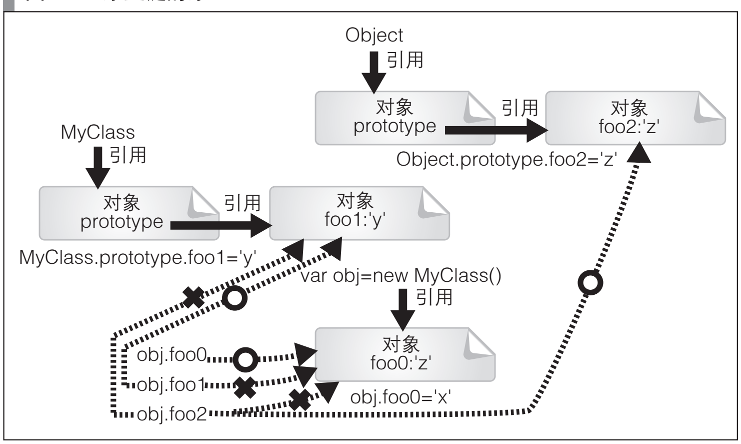
请注意，在读取和写入的时候，继承的执行方式是不同的，不过这种不对称性其实也是理所当然的。根据原型链的原理，所有的对象最终都会具有一个引用了 Object.prototype 对象的隐式链接。如果对属性的改写会影响到上一级对象的话，那么即使仅仅是改写了某一个对象的 toString 方法，也会对其他所有对象造成影响。这样一来就很难控制程序了。

另一方面，由于在读取中会发生继承，所以若是改写了某个隐式链接的 toString 方法，就能够在原型继承了该对象的对象中使用这一新的实现。这种对实现的继承或者说是对操作的继承的做法，正是对面向对象技术的一种灵活运用。

下面要介绍的术语可能会造成一些理解上的混乱：“隐式链接”所引用的对象被称为原型对象（请参见专栏）。在接受了这个术语之后，对于原型继承的说明就变得非常简单了。只需要通过“在读取属性时，对属性对象的属性进行继承”这样一句话就可以完成定义。

下面是对原型链的原理的图示（图 5.9）。请注意，变量和引用的对象是分开写的。这里要再次提醒的是，对象自身是没有名字的。如果记不清了，请重新回顾 5.2.3 节中的说明。

图 5.9 原型链的原理



5.16.2　原型链的具体示例

接下来，将说明原型链的具体示例以及其内部的执行方式。首先请看图 5.10。

图 5.10　原型链的具体示例（属性读取）

```
js> function MyClass() { this.x = 'x in MyClass'; }

js> var obj = new MyClass();    // 通过 MyClass 构造函数生成对象
js> print(obj.x);               // 访问对象 obj 的属性 x
x in MyClass

js> print(obj.z);               // 对象 obj 中没有属性 z
undefined

// Function 对象具有一个隐式的 prototype 属性
js> MyClass.prototype.z = 'z in MyClass.prototype';
                                // 在构造函数 prototype 对象新增属性 z
js> print(obj.z);               // 这里的 obj.z 访问的是构造函数 prototype 对象的属性
z in MyClass.prototype
```

在读取对象 obj 的属性的时候，将首先查找自身的属性。如果没有找到，则会进一步查找对象 MyClass 的 prototype 对象的属性。这就是原型链的基本原理。这样一来，在通过 MyClass 构造函数生成的对象之间就实现了对 MyClass.prototype 对象的属性的共享。

这种共享用面向对象的术语来说就是继承。通过继承可以生成具有同样执行方式的对象。不过请注意，在上面的代码中，如果修改 MyClass.prototype，已经生成的对象也会发生相应的变化。

而属性的写入与删除则与原型链无关。请看图 5.11、图 5.12 和图 5.13。

图 5.11　原型链的具体示例（属性写入）

```
js> function MyClass() { this.x = 'x in MyClass'; }
js> MyClass.prototype.y = 'y in MyClass.prototype';

js> var obj = new MyClass();    // 通过 MyClass 构造函数来生成对象
js> print(obj.y);               // 通过原型链读取属性
y in MyClass.prototype

js> obj.y = 'override';         // 在对象 obj 中新增直接属性 y
js> print(obj.y);               // 读取直接属性
'override'

js> var obj2 = new MyClass();
js> print(obj2.y);              // 在其他的对象中，属性 y 不会发生变化
y in MyClass.prototype
```

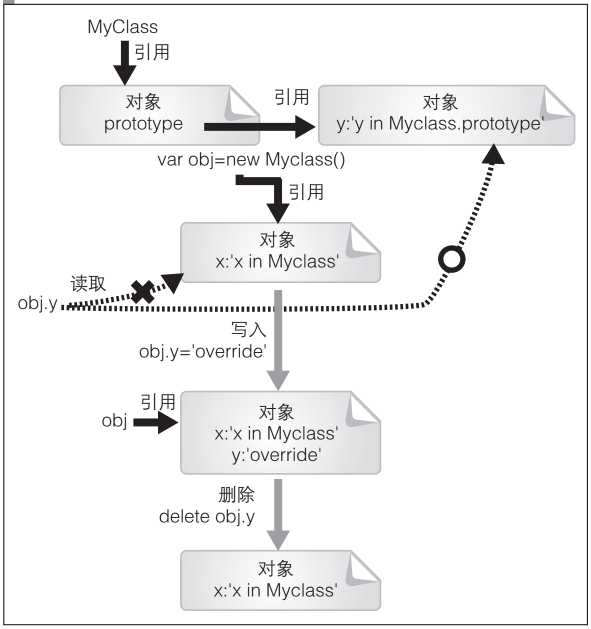
图 5.12　原型链的具体示例（属性删除）（接图 5.11）

```
js> delete obj.y;               // 删除属性 y

js> print(obj.y);               // 该直接属性不存在，因此将搜索原型链
y in MyClass.prototype

js> delete obj.y;               // 虽然 delete 运算的值为 true……
true
js> print(obj.y);               // 但无法 delete 原型链中的属性
y in MyClass.prototype
```

图 5.13 图 5.11 与图 5.12 的执行方式



5.16.3 原型继承与类

在图 5.10 中，如果没能找到 MyClass.prototype 对象的属性的话，则会继续搜索原型链。

之后将会在生成了 MyClass.prototype 对象的构造函数的 prototype 对象的属性中搜索。在默认情况下，MyClass.prototype 对象的构造函数是一个 Object 对象。因此，将会查找 Object.prototype 对象的属性。可以通过向 Object.prototype 对象增加新的属性来验证这一操作。不过如果在实际的代码中修改 Object.prototype 的话，将会对程序造成巨大的影响，所以并不推荐这么做。toString 方法是原本就存在于 Object.prototype 对象中的一个属性，可以通过对它调用来验证上面的说法（图 5.14）。可以通过 hasOwnProperty 方法来确认一个属性是否直接属于某个对象。需要注意的是，toString 并不是 Object 对象所具有的属性，而是 Object.prototype 对象的属性。

图 5.14 对是否存在一个指向 Object.prototype 的隐式链接的确认（接图 5.10）

```
js> obj.toString();                        // 检验是否可以对对象 obj 调用 toString 方法
[object Object]

js> obj.hasOwnProperty('toString');       // 在对象 obj 中不存在 toString 方法
false
js> Object.prototype.hasOwnProperty('toString');  // 在 Object.prototype 对象中存在 toString 方法
true

js> Object.hasOwnProperty('toString');    // 注意，在 Object 中并不存在 toString 方法
false
```

借助原型继承，在 JavaScript 中实现了类似于 Java 与 C++ 等基于类的程序设计语言中的类型层级的机制。

5.16.4 对于原型链的常见误解以及 _proto_ 属性

对于原型链，有以下常见的错误理解。

- **在搜索了自身的属性之后，将会查找构造函数自身的属性（对于图 5.10 中的例子来说，指的就是查找 MyClass.z）。**
- **在搜索了自身的属性之后，将查找对象的 prototype 对象的属性（对于图 5.10 中的例子来说，指的就是查找 obj.prototype.z）。**

原型链最终是通过"隐式链接"连接而成的。在一些 JavaScript 实现中具有 _proto_ 这样一个属性，它指向了隐式链接所引用的对象。

不过在 ECMAScript 的标准中并没有 _proto_ 属性，所以是否可以使用该属性还要取决于具体的实现。

5.16.5　原型对象

对象的隐式链接（_proto_ 属性）所引用的对象称为原型对象。下面的代码将说明这种命名方式可能会引起的问题。

```
function MyClass() {}
var obj = new MyClass();
```

MyClass.prototype 与 obj._proto_ 引用了同一个对象，即对象 obj 的原型对象。不过容易弄错的一点是，MyClass.prototype 的引用对象并不是 MyClass 的原型对象。（那么 MyClass 对象的原型对象是什么呢？答案是 Function.prototype 所引用的对象。更为详细的内容，请参见 6.6.1 节）。

在 5.2.3 节中，为方便起见，我们将变量 obj 所引用的对象称作"对象 obj"。但本节并没有使用"MyClass.prototype 对象"这样的称法。因为这样的话将会使术语的使用变得更加混乱。有时会将对象以对其引用的变量来命名，这种做法将会使本应不具有名字对象看起来好像是有名字似的，从而导致表述上的混乱。

5.16.6　ECMAScript 第 5 版与原型对象

原型链之所以不容易被理解，其重要原因之一是由于隐式链接的存在。在之前的小节中所介绍的 _proto_ 属性是非通用的增强功能，因此，实际上并没有一种可以从一个对象追溯至其原型对象的官方方法。这也是为什么隐式链接会被称为隐式链接。

这种情况在 ECMAScript 第 5 版中得到了改善。在 ECMAScript 第 5 版中有 getPrototypeOf 这样一个方法，它将会返回"隐式链接"所引用的对象。也就是说，在官方的标准中出现了一个和 _proto_ 属性这一非通用功能的作用相同的方法。代码清单 5.12 介绍了从一个对象中直接获取其原型对象的具体方法。

代码清单 5.12　获取原型对象的三种方法

```
// 前提条件
function MyClass() {}
var Proto = MyClass.prototype;
var obj = new MyClass();   // 对象 obj 的原型对象是对象 proto

// 通过对象实例取得（ECMAScript 第 5 版中最直接的方法）
var Proto = Object.getPrototypeOf(obj);

// 通过对象实例取得（使用 _proto_ 属性这一增强功能）
var Proto = obj._proto_

// 通过对象实例以及其构造函数取得（无法确保总是有效）
var Proto = obj.constructor.prototype;
```

虽然 _proto_ 属性是一种非通用的功能，不过由于它更为直观且易于理解，因此接下来，我们还是以它为基础进行说明。如果一定要遵循标准的话，只需将使用了 _proto_ 的代码部分替换为 Object.getPrototypeOf 即可。

5.17 对象与数据类型

对于基于类的程序设计语言，对象的类型是由作为模型的类以及其对其进行实现的接口共同决定的。而在 JavaScript 中，不存在这种意义上的对象类型的概念。这是因为在 JavaScript 中根本就不存在类与接口的概念。不过从原理上来说，对象类型的概念与对象的操作这一概念存在不少共性，由此，也可以认为在 JavaScript 中其实是存在对象类型的概念的。

首先需要说明的是如何判断明确的对象类型，也就是判断基本类型。这种判断可以通过 typeof 运算符实现。由于已经介绍过相关内容，因此在此不再举例说明。

对于 Object 类型，typeof 运算的结果是字符串值 "object"。

JavaScript 的语言规范没有对 Object 类型进一步细分。开发者可以根据自己的需要，设计出任意的具有共同执行方式的对象。面向对象技术的核心思想之一就是要考虑对象的操作，所以在面向对象程序设计的过程中必须重视这个问题。

5.17.1 数据类型判定（constructor 属性）

可以通过使用对象的 constructor 属性来从对象处获取其构造函数。如果能获知对象的构造函数，也就能够知道该对象的原型继承情况了，于是便可以了解这个对象的一部分操作。虽然 JavaScript 并不适合以基于类的观点来分析，不过可以姑且认为图 5.15 中代码的含义是对对象的类进行确认。

图 5.15 通过 constructor 属性判断类型的例子

```
js> var d = new Date();
js> d.constructor;                    // 对象 d 的 constructor 属性引用了 Date
function Date() {
    [native code]
}
js> var arr = [1,2,3];
js> arr.constructor;                  // 对象 arr 的 constructor 属性引用了 Arry
function Array() {
    [native code]
}
js> var obj = {};
js> obj.constructor;                  // 通过字面量生成的对象的 constructor 属性引用了 Object
function Object() {
    [native code]
}
```

5.17.2 constructor 属性的注意点

constructor 属性不是对象的直接属性，而是通过原型链查找到的属性。因此，下面这段模拟了派生继承的代码，实现了相当于基类 Base 的功能。不过需要注意的是，它并不一定总能按照设想的方式执行功能。

```
js> function Derived() {}            // 该构造函数相当于派生类
js> function Base() {}               // 该构造函数相当于基类
js> Derived.prototype = new Base();

js> var obj = new Derived();         // 通过 Derived 构造函数生成 object 对象
js> obj.constructor;                 // 引用了对象 obj 的 constructor 属性引用了 Base
function Base() {
}
```

与 obj.constructor 的原型链相连的实体是 Derived.prototype.constructor。所以只需像下面这样对其进行显式的修改，就可以使这类派生继承获得期望的结果。

```
js> Derived.prototype.constructor = Derived;

js> obj.constructor;                    // 对象 obj 的 constructor 属性引用了 Derived
function Derived() {
}
```

5.17.3　数据类型判定（instance 运算与 isPrototypeOf 方法）

虽然也可以通过 constructor 属性来判断对象类型，不过更为常见的做法是使用 instanceof 运算来进行判断。具体方法为在运算符左侧书写对象引用，在右侧书写相应的构造函数。如果对象是通过右侧的构造函数生成的，则运算结果为真。对于通过原型链进行派生继承的情况，instanceof 运算也是有效的。

图 5.16 是一个具体的例子。

图 5.16　通过 instanceof 运算来判断类型的例子

```
js> var d = new Date();                 // 通过 Date 构造函数生成对象 d
js> d instanceof Date;
true
js> d instanceof Object;
true

js> function Derived() {}               // 该构造函数相当于派生类
js> function Base() {}                  // 该构造函数相当于基类
js> Derived.prototype = new Base();

js> var obj = new Derived();
js> obj instanceof Derived;
true
js> obj instanceof Base;
true
js> obj instanceof Object;
true
```

可以通过 Object 类的 isPrototypeOf 方法来确认原型对象，该方法将搜索原型链。图 5.17 是个具体例子。

图 5.17　isPrototypeOf 方法的例子（接图 5.16）

```
js> Derived.prototype.isPrototypeOf(obj);
true
js> Base.prototype,isPrototypeOf(obj);
true
js> Object.prototype.isPrototypeOf(obj);
true
```

5.17.4　数据类型判定（鸭子类型）

如果类与对象（实例）之间只是静态关系，只需通过 instanceof 运算就能够解决所有的对象类型判断问题。然而，在 JavaScript 中，对象是一种动态的概念。例如，像下面这样，为生成的对象新增属性或不通过构造函数生成对象，都很常见。

```
js> var obj = {}                                  // 生成空对象
js> obj.doit = function() { print('doit'); }      // 新增属性

// 不通过构造函数生成对象
js> var obj = { doit: function() { print('doit'); } }
```

在上面这个对象中有一个名为 doit 的方法。对象所拥有的方法是用于描述对象操作的重要指标，然而一个方法是否存在却是无法通过 constructor 属性或 instanceof 运算来判断的。

判断在对象中含有哪些属性，是比 instanceof 运算更为普遍的类型判断方式。这种直接分析对象的操

作以判断其类型的方法俗称为鸭子类型判断。

in运算是一种可以用于判断鸭子类型的方法。in运算需要在运算符左侧书写属性名字符串，在右侧指定对象的引用。如果对象拥有所指定的属性，运算结果即为真。对于通过原型链继承的属性，也能够通过这一方式判断。

```
js> var obj = { doit: function() { print('doit'); } }
js> 'doit' in obj;              // 对象obj中含有doit属性，所以结果为真
true
js> 'toString' in obj;          // 从Object中继承了toString属性，所以结果为真
true
```

5.17.5 属性的枚举（原型继承的相关问题）

5.8.3节和5.9.1节已经介绍了通过for in语句以及for each in语句来枚举属性。

此外，前面的小节还介绍了通过in运算符来判断属性是否存在。for in语句、for each in语句以及in语句都会对原型链进行搜索。对于类型判断来说，搜索原型链是一种方便的做法。不过，我们有时仅希望对直接属性是否存在进行判断，并不需要对原型链进行搜索。这时，可以使用hasOwnProperty方法。下面是个具体例子。

```
// 仅列举直接属性的代码示例
for (var key in obj) {
    if (obj.hasOwnProperty(key)) {
        print(key);
    }
}
```

5.18节中的keys方法和getOwnPropertyNames方法会返回一个数组，它是由作为参数传递而来的对象的直接属性的名称组成的。其中key方法仅返回enumerable属性为真的属性，它得到的结果与通过for in语句中的hasOwnProperty方法判断得到的是相同的。而getOwnPropertyNames方法则是在返回属性名时无视enumerable属性。

下面是Object类的keys方法与getOwnPropertyNames方法的具体例子。

```
js> var obj = { x:1, y:2 };
js> Object.keys(obj);
["x", "y"]
js> Object.getOwnPropertyNames(obj);
["x", "y"]

// 数组也是一种对象，详细内容请参见7.1节
js> var arr = [3,4];
js> Object.keys(arr);
["0", "1"]
js> Object.getOwnPropertyNames(arr);
["length", "0", "1"]

// 对Object.prototype对象的考察
js> Object.keys(Object.prototype);
[]
js> Object.getOwnPropertyNames(Object.prototype);
["constructor", "toSource", "toString", "toLocaleString", "valueOf", "watch",
"unwatch", "hasOwnProperty", "isPrototypeOf", "propertyIsEnumerable", "_defineGetter_",
"_defineSetter_", "_lookupGetter_", "_lookupSetter_"]
```

顺带一提，对于属性的enumerable属性可以通过propertyIsEnumerable方法来判断。

5.18 ECMAScript 第 5 版中的 Object 类

ECMAScript 第 5 版中 Object 类的 create 方法，是除了对象字面量与 new 表达式之外的第三种官方的对象生成方法。它的第一个参数需要一个原型对象，而第二个参数需要一个属性对象（之后将作说明）。

如果将一个 null 作为原型对象传递给 create 方法，则会生成一个没有进行原型继承的对象。请看下面的例子。

```
// 未原型继承于 Object 类的对象

js> var obj = Object.create(null);
js> print(Object.getPrototypeOf(obj));
null
js> 'toString' in obj;                      // 对没有继承 toString 进行确认
false
```

可以用下面的代码来实现与通过对象字面量生成对象相同的效果。

```
js> var obj = Object.create(Object.prototype);   // 与 var obj = {} 等价
```

通过使用 Object.create 方法，可以在代码中更为直观地表述原型继承。代码清单 5.13 是一个具体的例子。

代码清单 5.13　Object.create 方法的示例

```
function MyClass() {}
var Proto = { x:2, y:3 };                   // 原型对象
MyClass.prototype = Proto;
var obj = new MyClass();

与下面的代码等价

var Proto = { x:2, y:3 };                   // 原型对象
var obj = Object.creat(Proto);
```

```
// 运行代码清单 5.13 的示例
js> print(obj.x, obj.y);                    // 确认属性已被原型继承
2 3
```

5.18.1　属性对象

create 方法的第二个参数是一个关联数组，其键为属性名，其值为属性描述符（属性对象）。属性描述符指的是在表 5.1 中的由属性的属性组成的关联数组。

下面是个具体例子。其中属性值是通过 value 属性指定的。大部分属性的默认值是 false，在这个例子中会将它们显式地指定为 true。

```
js> var obj = { x:2, y:3 };
与下面的代码等价
js> var obj = Object.create(Object.prototype,
              { x: {value:2, writable:true, enumerable:true, configurable:true},
                y: {value:3, writable:true, enumerable:true, configurable:true} });
```

可以通过 Object.create 来显式地指定属性的属性。表 5.4 列出了与属性的属性相关的方法。

表 5.4　与 Object 类的属性的属性有关的方法（表 5.1 的摘选）

方法	说明
defineProperty(o, p, attributes)	向对象 o 增加 / 更新具有特定信息的属性 p
defineProperties(o, properties)	向对象 o 增加 / 更新具有特定信息的属性
getOwnPropertyDescriptor(o, p)	返回对象 o 的直接属性 p 的信息（值与属性）

下面是表 5.4 中的方法的一些具体示例。

```
js> var obj = Object.create(Object.prototype, { x:{value:2} });

// 除了显式指定的属性，其他的值都为 false（value 的默认值为 undefined）
js> Object.getOwnPropertyDescriptor(obj, 'x');
({value:2, writable:false, enumerable:false, configurable,false})

// 新增属性 y
js> Object.defineProperty(obj, 'y', {value:3, enumerable:true});

js> Object.getOwnPropertyDescriptor(obj, 'y');       // 确认
({value:3, writable:false, enumerable:true, configurable,false})


// 新增属性 z
js> Object.defineProperties(obj, { z:{value:function(){ print('z called'); },
    enumerable:true } });

js Object.getOwnPropertyDescriptor(obj, 'z');        // 确认
({value:(function () {print("z called");}), writable:false, enumerable:true,
    configurable:false})

// 确认 enumerable 属性（也可以通过 Object.keys 实现）
js> for (var key in obj) {
         print(key);
    }
y
z
```

如果属性的 configurable 属性为 true，则可以更改包括值在内的所有属性，反之如果为 false，则不能。由于此时 configurable 属性也无法被更改，因此事实上如果该属性为 false，则将无法进行任何更改。

5.18.2 访问器的属性

只要将 get 属性与 set 属性指定为相应的函数，就能够定义一个只能够通过访问器 getter 和 setter 来访问值的属性。访问器与 value 属性是相互排斥的，也就是说，如果指定了 value 属性的值，访问器（同时包括 get 和 set）就会失效；反之，如果指定了访问器（get 或 set 中的某一个），value 属性就会失效。

对 get 属性需要指定一个有返回值的函数，对于 set 属性则需要指定一个能够接受一个参数并在内部变更状态的函数。从内部来看，将属性作为右值访问时使用的是 getter 函数，而将属性作为左值以进行赋值时使用的是 setter 函数。根据语言规则，是可以把提供其他功能的函数指定给 get 属性的，不过那样就没有意义了，所以还是应该使用功能相符的函数。图 5.18 中的代码验证了访问器的功能。只要能写出正确的 getter 访问器函数，就能以此为基础设计出一个不可变对象。

图 5.18 对访问器的功能的验证

```
js> var obj = Object.create(Object.prototype,
         { x:{ get:function(){ print('get called');},
               set:function(){ print('set called');}
               }
         });

js> print(obj.x);      // 当要读取属性时，将会调用 getter 函数
get called
undefined    // 返回 getter 函数的返回值（由于上面的代码中没有 return 任何值，因此结果为 undefined 值

js> obj.x = 1;         // 在写入属性值时将调用 setter 函数
set called

js> print(obj.x);      // 如上所示，此处的 setter 函数不会对属性做任何更改
get called
```

```
undefined
```

访问器函数也可以通过对象字面量来表述。下面代码中的对象和图 5.18 中的是一样的。

```
var obj = { get x() { print('get called'); },
            set x(v) { print('set called'); } };
```

getter 函数与 setter 函数中的 this 引用指向的是当前对象，不过下面这样的代码是无法运行的。这是因为其中的每个访问器函数都会不断调用访问器函数。

```
// 无法运行（有无限循环的致命错误）

var obj = Object.create(Object.prototype,
              { x:{ get:function(){ return this.x; },
                    set:function(v){ this.x = v; }
                  }
              });
```

下面的代码虽然能够运行，但还是有些问题。

```
// 使用了隐藏的属性 _x 的访问器的例子（姑且算是能够运行）

var obj = Object.create(Object.prototype,
              { x:{ get:function(){ return this._x; },
                    set:function(v){ this._x = v;} },
                x:{ writable:true }
              });
```

这段代码的问题之处在于属性 _x 是可以从外部改写的。如果通过代码规范，规定不允许从外部直接访问这一属性的话倒是可以解决问题，不过这种规范通常很有可能会被打破。不借助于规范的更恰当的方法是通过闭包来隐藏这个变量。之后的 6.7.5 节将对此进行说明。代码清单 5.14 是一个具体示例。

代码清单 5.14　闭包与访问器相结合

```
// 用于生成对象的函数（参见 5.7.1 节 ）
function createObject() {
    var _x = 0;       // 变量名也可以用 x，不过那样容易产生混乱，所以这里仍使用 _x

// 返回一个定义了访问器的对象
return { get x() { return _x; },
         set x(v) { _x = v; }
       };
}
```

```
// 调用代码清单 5.14 中的方法的示例
js> var obj = createObject();  // 生成对象

js> print(obj.x);              // 读取（在内部调用 getter）
0
js> obj.x = 1;                 // 改写（在内部调用 setter）
1
js> print(obj.x);
1
```

在代码清单 5.14 使用的是对象字面量，不过其实也可以通过使用 Object.create 以及 Object.defineProperty 方法来实现同样的效果。

专栏

其他种类的类型判断

ECMAScript 第 5 版提供了 Array.isArray 方法。建议在判断数组类型时使用该方法。

本节介绍了一些判断类型的方法，而在 jQuery 或 prototype.js 等知名的库中，类型判断是基于字符串实

现的。也就是说，将会先通过 toString 使对象字符串化，之后再对其进行模式匹配。尽管不能算是一种优美的方式，不过这是一种在实际开发中切实可行的库解决方案。

5.19 标准对象

下表（表 5.5）是在 ECMAScript 第 5 版中定义的标准内建对象（built-in object）。其中的一些对象是以类的形式来表述的，这是因为将它们视为类的话将会更易于理解。关于如何使用相关的术语，请参见 2.5.7 节。

表 5.5 ECMAScript 第 5 版中的内建对象

名称	说明
Object	所有对象的基类
（通称）全局对象	该对象的属性是全局变量或全局函数
String	字符串类
Array	数组类
Function	函数类
Number	数值类
Boolean	布尔类
Math	数学函数对象
Date	日期类
RegExp	正则表达式类
JSON	JSON 解释器类
Error	错误基类
EvalError	求值错误类
RangeError	越界错误类
ReferenceError	引用错误类
SystaxError	语法错误类
TypeError	类型错误类
URIError	URI 错误类

5.20 Object 类

Object 类是 JavaScript 中所有的类的基类[①]。其名称与作用都与 Java 的 Object 类类似，不过其继承机制与 Java 不同，采用了原型继承的方式。

如下面的具体示例所示，在 Object.prototype 对象中增加的属性能够在任意对象中进行读取访问。不过这里仅仅是为了说明而向 Object.prototype 对象新增了属性，实际开发中如无必要不应该这样做，这种做法的影响范围过于巨大，容易引发错误。

```
js> Object.prototype.foobar = 'FOOBAR'; // 在 Object.prototype 对象中新增属性
FOOBAR

js> var d = new Date();                 // 可以取任意的对象，这里以 Date 对象为例
js> d.foobar;                           // 读取 foobar 属性就能够得到之前的值
FOOBAR

js> 'x'.foobar;                         // 对于字符串对象也能够读取 foobar 属性
FOOBAR

js> (0).foobar;                         // 对于数值对象也能够读取 foobar 属性
```

① 也可以生成没有（原型）继承 Object 类的对象（请参见 5.18 节）。

FOOBAR

表 5.6 总结了 Object 类的函数以及构造函数调用。

表 5.6　Object 类的函数以及构造函数调用

函数或者构造函数	说明
Object()	生成 Object 实例
Object(value)	将参数 value 转换为 Object 对象并生成 Object 实例
new Object()	生成 Object 实例
new Object(value)	将参数 value 转换为 Object 对象并生成 Object 实例

在没有特别理由的情况下，建议通过字面量来生成对象，而不要使用 Object 类的函数或构造函数调用。

表 5.7 是 Object 类的属性。可以以形如 Object.seal(obj) 的方式使用。

表 5.7　Object 类的属性

属性名	说明
create(o, [properties])	以对象 o 为原型并返回具有指定属性的实例
defineProperty(o, p, attributes)	向对象 o 增加 / 更新具有特定信息的属性 p
defineProperties(o, properties)	向对象 o 增加 / 更新具有特定信息的属性
freeze(o)	参见 5.12 节
getPrototypeOf(o)	返回对象 o 的原型对象
getOwnPropertyDescriptor(o, p)	返回对象 o 的直接属性 p 的信息（值与属性）
getOwnPropertyNames(o)	返回对象 o 的直接属性名组成的数组
isSealed(o)	参见 5.12 节
isFrozen(o)	参见 5.12 节
isExtensible(o)	参见 5.12 节
keys(o)	返回对象 o 以及继承的属性名组成的数组
length	值为 1
preventExtensions(o)	参见 5.12 节
prototype	用于原型链
seal(o)	参见 5.12 节

表 5.8 总结了 Object.prototype 对象的属性。

表 5.8　Object.prototype 对象的属性

属性名	说明
constructor	Object 类对象的引用
hasOwnProperty(v)	如果字符串 v 是实例的直接属性名，则返回真
isPrototypeOf(v)	如果对象 v 是实例的原型，则返回真
propertyIsEnumerable(v)	如果字符串 v 是实例中可枚举的属性名，则返回真
toSource()	JavaScript 的增强功能。其求值结果将返回用于生成实例的字符串
toLocaleString()	将实例转换为与位置相关的字符串值。一般由开发者根据需要实现
toString()	将实例转换为字符串值。一般由开发者根据需要实现
unwatch(p)	JavaScript 的增强功能。删除属性 p 的观察点
valueOf()	将实例转换为恰当的值。如有必要，由开发者实现
watch(p, handle)	JavaScript 的增强功能。对属性 p 设置观察点（一个会在值发生改变时被调用的函数）
defineGetter(p, getter)	JavaScript 的增强功能。对属性 p 设置 getter 属性（*1）
defineSetter(p, setter)	JavaScript 的增强功能。对属性 p 设置 setter 属性（*1）
lookupGetter(p)	JavaScript 的增强功能。返回属性 p 的 getter 属性（*1）
lookupSetter(p)	JavaScript 的增强功能。返回属性 p 的 setter 属性（*1）
noSuchMethod	JavaScript 的增强功能。如果对对象调用了不存在的方法，该挂钩函数将会被调用（*2）
proto	JavaScript 的增强功能。（*3）

*1 参见 5.18 节
*2 这个函数相当于 Ruby 中的 method_missing，这么说或许有些读者就能立即理解了
*3 参见 5.16.4 节

Object 类没有实例属性。关于 Object 类的详细信息，请参见 5.12 节。

5.21 全局对象

全局对象相当于宿主对象的根对象。关于宿主对象的内容，请参见第 1 章。可以通过在最外层代码中使用 this 引用来访问全局对象。

专栏

对象的兼容性

可以通过辅助性的代码来解决属性之间不兼容问题。在 JavaScript 中，开发者在之后可以自由地向标准对象添加属性。例如，如果希望向 String 对象添加一个 trim 方法用以删除字符串前后的空白字符，可以通过以下方式实现。如果 trim 方法已经存在，则不会产生任何影响。关于向 String.prototype 新增方法的详细内容，请参见 5.16 节。

```
// 用以提高兼容性的代码示例

if (!String.prototype.trim) {
    String.prototype.trim = function() {
        return String(this).replace(/^\s+|\s+$/g, "");
    };
}
```

能够变更标准对象的属性是一回事，而是否应该变更则是另一回事。这涉及对问题的思考方式的差异，并不能一概而论。

在表 5.5 中，除了全局对象外，还有一些被称为 Object 或 String 等的对象。这里对此简单做一下说明。已经反复强调过，对象自身是不具有名称的。这一点对于 Object 对象也好，对于 String 对象也好，都是不变的，这些对象本身没有名字，只不过是可以通过 Object 或 String 这样的名称访问而已。

对于全局对象而言，JavaScript 并没有从语言标准上规定其名称。在客户端 JavaScript 中预先存在一个指向了全局对象的变量 window，对此本书的第 3 部分也会再次说明。因此，如同 Object 对象以及 String 对象，全局对象也被称为 window 对象[①]。不过要注意 window 变量仅存在于客户端 JavaScript 中，在 ECMAScript 的核心语言标准中，全局变量并没有特定的引用名称。因为在当前这一部分中将仅说明核心语言，所以就没有对全局变量采用专门的名称，而直接称其为全局变量。

不过由于最外层代码中的 this 引用指向了全局变量，因此只要通过下面这样的代码，就可以在任何环境中实现以变量 global 对全局变量的引用。在客户端 JavaScript 中这并不是必需的（因为已经有了 window 这一变量名称），而在包括服务器端 JavaScript 在内的其他环境中，通过这种方式可以统一命名，是一种合理的做法。

```
// 可以通过在最外层代码中使用下面的代码来实现在整个程序中通过 global 对全局对象的引用
var global = this;
```

5.21.1 全局对象与全局变量

正如在 5.3 节中所说，全局变量与全局函数是全局对象的属性。也就是说，Object 或 String 这些名字也

① 根据 5.2.3 节说明，是应该使用对象 window 或是对象 Object 对象这样的称法的，不过在这里由于考虑了自然易用性而对术语使用方式的一贯性做了一些妥协。

都是全局对象的属性名。熟悉了 Java 这样的具有类型名称的读者可能会觉得奇怪，为什么在 JavaScript 中类型名称都是属性名（变量名），或者说是否在 JavaScript 中根本就没有类型名称这样的概念。在前一节中提到的变量名 global 以及 window 自然也都是全局对象的属性名。乍一看，属性名可以引用包含了自身的全局对象，是一种容易产生混乱的关系，不过这一点也已经图 5.4 已经做了说明。

根据运行环境的不同，全局对象中含有的属性也不相同。ECMAScript 第 5 版规定的属性是符合标准的最小集，在通常情况下都会具有更多的属性。例如，对于客户端 JavaScript 的情况，DOM 对象是事先存在的。更为详细的内容将在本书第 3 部分说明。

表 5.9 总结了 ECMAScript 第 5 版中全局对象的属性[①]。注意，无法对全局对象进行函数调用或构造函数调用。

表 5.9　ECMAScript 第 5 版中全局对象的属性

属性名	说明
NaN	表示 Not a Number 含义的值
Infinity	表示无穷大的值
undefined	表示 undefined 类型的值
eval(x)	将参数的字符串值 x 作为 JavaScript 代码求值（执行）
parseInt(str, radix)	将字符串值 str 转换为基数为 radix 的整数值
parseFloat(str)	将字符串值 str 转换为数值
isNaN(num)	如果数值 num 是 NaN 的话则返回真
isFinite(num)	如果数值 num 是三种特殊值（NaN 以及正负无穷大）的话则为真，否则为假
encodeURI(uri)	将字符串值 uri 编码为 URI 字符串值，并返回其中除了 URI 特殊字符（？或 & 之类）以外的部分。
decodeURI(encodedURI)	encodeURI 函数的逆向转换
encodeURIComponent(uriComponent)	将字符串值 rui 编码为 URI 字符串值并返回
decodeURIComponent(encodedURIComponent)	encodeURIComponent 函数的逆向转换

5.21.2　Math 对象

Math 对象是提供了数学函数等功能的对象，无法对其进行构造函数调用。用 Java 的术语来说的话，这就相当于一个直接调用类方法的工具类。

表 5.10 总结了 Math 对象的属性。它们可以以 Math.random() 的形式来使用。

表 5.10　Math 对象的属性

属性名	说明
E	自然对数的底（2.7182818284590452354）
LN2	2 的自然对数（0.6931471805599453）
LN10	10 的自然对数（2.302585092994046）
LOG2E	以 2 为底的 E 的对数（1.4426950408889634）
LOG10E	以 10 为底的 E 的对数（0.4342944819032518）
PI	圆周率（3.1415926535897932）
SQRT1_2	1/2 的平方根（0.7071067811865476）
SQRT2	2 的平方根（1.4142135623730951）
abs(x)	x 的绝对值
acos(x)	x 的 arccos
asin(x)	x 的 arcsin
atan(x)	x 的 arctan
atan2(y, x)	y/x 的 arctan（坐标 x,y 的弧度制角度）
ceil(x)	大于等于 x 的最小整数
cos(x)	x 的 cos

① 标准内建对象的名称（Object 或 String 等）也应该是表 5.5 内容的一部分，不过在此暂且省略。

（续）

属性名	说明
exp(x)	e 的 x 次方
floor(x)	小于等于 x 的最大整数
log(x)	x 的自然对数（底为 e）
max([value0, [value1, value2, ...]])	参数中的最大值
min([value0, [value1, value2, ...]])	参数中的最小值
pox(x, y)	x 的 y 次方
random()	大于等于 0 小于 1 的随机数
round(x)	x 四舍五入后的整数
sin(x)	x 的 sin
sqrt(x)	x 的平方根
tan(x)	x 的 tan
toSource	JavaScript 自带的增强功能。返回字符串 "Math"

5.21.3 Error 对象

表 5.11 总结了 Error 类的函数以及构造函数调用，而表 5.12 则总结了 Error 类的属性。

表 5.11 Error 类的函数以及构造函数调用

函数或构造函数	说明
Error(message)	生成一个 Error 实例
new Error(message)	生成一个 Error 实例
Error(message, fileName, lineNumber)	JavaScript 自带的增强功能。生成一个 Error 实例
new Error(message, fileName, lineNumer)	JavaScript 自带的增强功能。生成一个 Error 实例

表 5.12 Error 类的属性

属性名	说明
length	值为 1
prototype	用于原型链

表 5.13 总结了 Error.prototype 对象的属性。

表 5.13 Error.prototype 对象的属性

属性名	说明
constructor	对 String 类对象的引用
message	错误信息
name	表示错误类型的字符串。例如，是 EvalError 的话则是 "EvalError"，是 RangeError 的话则是 "RangeError" 等
fileName	JavaScript 自带的增强功能。发生错误的文件名
lineNumber	JavaScript 自带的增强功能。发生错误的行号
stack	JavaScript 自带的增强功能。发生错误时的调用栈
toSource()	JavaScript 自带的增强功能。其求值结果将返回生成了这一 Error 实例的字符串
toString()	将 Error 实例转换为字符串值

表 5.5 中的其他的错误类都原型继承于 Error 类。这一点可以通过以下的方式验证。

```
js> Error.prototype._proto_ === Object.prototype;        // Error 继承于 Object
true

js> EvalError.prototype._proto_ === Error.prototype;     // EvalError 继承于 Error
true
```

第 6 章　函数与闭包

本章讲解函数与闭包。借助函数，就能够使用子程序，不过仅仅使用函数还称不上真正发挥了 JavaScript 语言的力量。应加深对“函数本身也可以是运算对象”的认识，同时充分理解闭包的机制，从而跨入函数式编程的世界。

6.1　函数声明语句与匿名函数表达式

可以通过函数声明语句与匿名函数表达式对函数进行声明。关于声明的语法格式，请参见 2.4 节。通过函数声明语句声明的函数在声明之前就可以调用（参见之后的 6.2.1 节）。

6.2　函数调用的分类

表 6.1 对函数调用方式之间的区别进行了说明。请结合 5.14 节中的表 5.3 使用。

表 6.1　函数调用的分类

名称	说明
方法调用	通过接收方对象对函数进行调用（包括 apply 与 call 调用）
构造函数调用	通过 new 表达式对函数进行调用
函数调用	以上两种方式之外的函数调用

请注意，这并不是对函数本身的分类，而是对不同的函数调用方式的分类。也就是说，将一个函数称为方法并不严谨。更确切的说法是，该函数是（或者不是）通过方法调用的方式使用的。话虽如此，过分拘泥于使用严谨的术语定义也显得有些死板，所以通常来说，将以方法调用的方式使用的函数称为方法，同理，将以构造函数调用方式使用的函数称为构造函数。

在下文中，基本上都会使用函数这一术语。不过，函数与方法或构造函数主要是名称上的差别，因此对函数的说明也适用于方法和构造函数。

函数声明语句的后置

通过函数声明语句声明的函数，可以在进行声明的代码行之前就对其调用。请看下面的具体示例。虽然这个例子在函数的作用域内进行，不过对于全局作用域情况也是相同的。

```
js> function doit() {
        fn();      // 在声明函数 fn 之前对其进行调用
        function fn() { print('called'); };
    }

// 函数调用
js> doit();
called
```

在通过匿名函数表达式进行定义的情况下结果将会不同。下面的代码结构上与上面的类似，却是错误的。

```
js> function doit() {
        fn();
        var fn = function() { print('called'); };
    }

// 函数调用
js> doit()
TypeError: fn is not a function
```

6.3 参数与局部变量

6.3.1 arguments 对象

可以通过在函数内使用 arguments 对象来访问实参。使用方式如代码清单 6.1 所示。

代码清单 6.1 使用 arguments 对象的例子

```
function fn() {
    print(arguments.length);
    print(arguments[0], arguments[1], arguments[2]);
}
```

```
// 对代码清单 6.1 的调用
js> fn(7);        // arguments.length 为实参的数量，值为 1。arguments[0] 的值为 7
1
7 undefined undefined

js> fn(7, 8);
// arguments.length 为实参的数量，值为 2。arguments[0] 的值为 7，arguments[1] 的值为 8
2
7 8 undefined

js> fn(7, 8, 9);
// arguments.length 为实参的数量，值为 1。arguments[0] 的值为 7，arguments[1] 的值为 8，arguments[2] 的值为 9
3
7 8 9
```

没有相对应的形参的实参也可以通过 arguments 访问。由于能够通过 arguments.length 获知实参的数量，因此可以写出所谓的可变长参数函数。而形参的数量则可以通过 Function 对象自身的 length 属性（在上面的例子中是 fn.length）来获得。

虽然 arguments 可以以数组的方式使用，不过它本身并不是数组对象。因此，无法对其使用数组类中的方法。详细内容请参见之后的 7.1.11 节。

6.3.2 递归函数

递归函数是一种在函数内对自身进行调用的函数。这种方式被称为递归执行或递归调用。代码清单 6.2 是个具体例子。

代码清单 6.2 n 的阶乘（递归函数的例子）

```
function factorial(n) {
    if (n <= 1 ) {
        return 1;
    } else {
```

```
        return n * factorial(n - 1);
    }
}
```

```
// 对代码清单 6.2 的调用
js> factorial(5);     // 5!=(5*4*3*2*1)=120
120
```

如果递归函数不停地调用自身，运行将不会终止（这与无限循环的情况是一样的，因此俗称为无限递归）。JavaScript 发生无限递归之后的反应取决于实际的运行环境。如果是 SpiderMonkey 的壳层，则会像下面这样发生 InternalError。而在 Java6 附带的 Rhino 中，发生无限递归后则会产生 java.lang.OutOfMemoryError 而使 Rhino 停止运行。

```
// SpiderMonkey 中的无限递归

js> function fn() { fn(); }
js> fn();
InternalError: too much recursion
```

必须在递归函数内部设置递归执行的停止条件判断，这称为终止条件。对于代码清单 6.2 中的情况，在函数开始处会对参数 n 的值是否小于等于 1 进行判断。终止条件的代码并不一定非要写在递归函数的头部，不过一般来说，写在头部更便于阅读。

可以通过循环实现的处理一定也能够通过递归处理来实现，反之也成立。这是因为，递归调用和循环处理两者的本质说到底都是反复执行某一操作。大多数情况下，通过循环来实现的代码会更为简洁明了。而且，在 JavaScript 中递归处理的执行效率并不一定很高。因此，一般情况下最好避免在 JavaScript 中使用递归。

能够通过 arguments.callee 来获取正在执行的 Function 对象的引用。这一引用可以在通过没有名字的函数（所谓的匿名函数）来实现递归函数时使用。下面是一个计算 n 的阶乘的具体示例（请注意，在 ECMAScript 第 5 版的静态模式中，arguments.callee 被禁止使用）。

```
// n 的阶乘（利用 arguments.callee）

js> (function(n) {is (n <= 1) return 1; else return n * arguments.callee(n - 1); })(5);
120
```

6.4 作用域

作用域指的是名称（变量名与函数名）的有效范围。关于作用域的有关内容，5.3 节和 5.4 节也有涉及。

在 JavaScript 中有以下两种作用域。

- **全局作用域**
- **函数作用域**

全局作用域是函数之外（最外层代码）的作用域。在函数之外进行声明的名称属于全局作用域。这些名称就是所谓的全局变量以及全局函数。

而在函数内进行声明的名称拥有的是函数作用域，它们仅在该函数内部才有效。相对于全局作用域，可以将其称为局部作用域；相对于全局变量，又可以将其称为局部变量。作为函数形参的参数变量也属于函数作用域。

JavaScript 的函数作用域的机制，与 Java（以及其他很多的程序设计语言）中的局部作用域有着微妙的差异。在 Java 中，局部变量所具有的作用域是从方法内对该变量进行声明的那一行开始的；而在 JavaScript 中，函数作用域与进行声明的行数没有关系。

请看代码清单 6.3 的例子。

代码清单 6.3 函数作用域的注意事项

```
var x = 1;
function f() {
    print('x = ' + x);         // 对变量 x 进行访问
    var x = 2;
    print('x = ' + x);         // 对变量 x 进行访问
}
```

```
// 对代码清单 6.3 的调用
js> f();
x = undefined
x = 2
```

乍一看，会认为函数 f 内的第一个 print 显示的是全局变量 x。然而，这里的 x 是在下一行进行声明的局部变量 x。这是因为，局部变量 x 的作用域是整个函数 f 内部。由于此时还没有对其进行赋值，因此变量 x 的值为 undefined 值。也就是说，函数 f 与下面的代码是等价的。

```
// 与代码清单 6.3 等价的代码

function f() {
    var x;
    print('x = ' + x);
    x = 2;
    print('x = ' + x);
}
```

代码清单 6.3 中的代码非常不易于理解，常常是发生错误的原因。因此，我们建议在函数的开始处对所有的局部变量进行声明。

Java 等语言建议直到要使用某一变量时才对其进行声明，不过 JavaScript 则有所不同，对此请加以注意。

6.4.1 浏览器与作用域

在客户端 JavaScript 中，各个窗口（标签）、框架（包括 iframe）都有其各自的全局作用域。在窗口之间是无法访问各自全局作用域中的名称的，但父辈与其框架之间可以相互访问。

更为详细的内容将在本书第 3 部分说明。

6.4.2 块级作用域

在 JavaScript（ECMAScript）中不存在块级作用域的概念，这一点与其他很多的程序设计语言不同。举例来说，请看图 6.1。如果认为块级作用域存在，就会认为第二个 print 的结果应该是 1，不过实际的输出却是 2。

图 6.1 对于块级作用域的误解

```
js> var x = 1;                         // 全局变量
js> { var x = 2; print('x = ' + x); }
x = 2;
js> print('x = ' + x);                 // 认为结果会是 1？
x = 2
```

在图 6.1 中，看似是在代码块内重新声明了块级作用域中的变量 x，但实际上，它只是将全局变量 x 赋值为了 2。也就是说，这与下面的代码是等价的。

```
// 与图 6.1 相等价的代码

js> var x = 1;                         // 全局变量
```

```
js> { x = 2; print('x = ' + x); }
x = 2
js> print('x = ' + x);
x = 2
```

在函数作用域中也存在这种对块级作用域的错误理解。在 for 语句中对循环变量进行声明是一种习惯做法，不过该循环变量的作用域并不局限于 for 语句内。在下面的代码中，其实是对局部变量 i 进行了循环使用。

```
function f() {
    var i = 1;
    for (var i = 0; i < 10; i++) {
        省略
    }
    此时变量 i 的值为 10
}
```

6.4.3　let 与块级作用域

虽然在 ECMAScript 第 5 版中没有块级作用域，不过 JavaScript 自带有 let 这一增强功能，可以实现块级作用域的效果。可以通过 let 定义（let 声明）、let 语句，以及 let 表达式三种方式来使用 let 功能。虽然语法结构不同，但是原理是一样的。接下来依次说明。

let 定义（let 声明）与 var 声明的用法相同。可以通过下面这样的语法结构对变量进行声明。

```
let var1 [= value1] [, var2 [= value2]] [, …, varN [= valueN]];
```

通过 let 声明进行声明的变量具有块级作用域。除了作用域之外，它的其他方面与通过 var 进行声明的变量没有区别。代码清单 6.4 中是个简单的例子。

代码清单 6.4　let 声明

```
function f() {
    let x = 1;
    print(x);                  // 输出 1
    {
        let x = 2;             // 输出 2
        print(x);
    }                          // let x = 2 的作用域到此为止
    print(x);                  // 输出 1
}
```

```
// 对代码清单 6.4 的调用
js> f();
1
2
1
```

如果不考虑作用域的不同，let 变量（通过 let 声明进行声明的变量）与 var 变量的执行方式非常相似。请参见代码清单 6.5 中的注释部分。

代码清单 6.5　let 变量的执行方式的具体示例

```
// 名称的查找
function f1() {
    let x = 1;
    {
        print(x);              // 输出 1。将对代码块由内至外进行名称查找
    }
}
```

```
// 该名称在进行 let 声明之前也是有效的
function f2() {
    let x = 1;
    {
        print(x);              // 这里是 let x = 2 的作用域。不过由于还未对其进行赋值，所以 let 变
                                  量 x 的值为 undefined
        let x = 2;
        print(x);              // 输出 2
    }
}
```

```
// 对代码清单 6.5 的调用
js> f1();
1

js> f2();undefined
2
```

像下面这样，将 for 语句的初始化表达式中的 var 声明改为 let 变量之后，作用域就将被限制于 for 语句之内。这样的做法更符合通常的思维方式。for in 语句以及 for each in 语句也是同理。

```
for (let i = 0, len = arr.length; i < len; i++) {
    print(arr[i]);
}
这里已是 let 变量 i 的作用域之外
```

let 语句的语法结构如下。let 变量的作用域被限制于语句内部。

```
let (var1 [= value1] [, var2 [= value2]] [, …, varN [= valueN]]) 语句;
```

下面是 let 语句的具体示例。

```
let (x = 1) {              // 代码块
    print(x);              // 输出 1
}                          // let 变量的作用域到此为止
```

代码清单 6.6 是一个混用 var 声明与 let 语句的具体示例。

代码清单 6.6　var 声明与 let 语句

```
function f() {
    var x = 1;
    let (x = 2) {
        print(x);          // 输出 2
        x = 3;
        print(x);          // 输出 3
    }
    print(x);              // 输出 1
}
```

```
// 对代码清单 6.6 的调用
js> f();
2
3
1
```

在 let 语句内部，声明与 let 变量同名的变量会引起 TypeError 问题。下面是一个例子。

```
// 不能通过 let 声明同名变量
let (x = 1) {
    let x = 2;
}
TypeError: redeclaration of variable x:


// 也不能通过 var 声明同名变量
```

```
let (x = 1) {
    var x = 2;
}
TypeError: redeclaration of let x:
```

let 表达式的语法结构如下所示。let 变量的作用域被限制于表达式内部。

```
let (var1 [= value1] [, var2 [= value2]] [, …, varN [= valueN]]) 表达式;
```

下面是 let 表达式的具体示例。

```
js> var x = 1;
js> var y = let(x = 2) x + 1;       // 在表达式 x+1 中使用了 let 变量(值为 2)
js> print(x, y);                    // 对 var 变量 x 的值没有影响
1 3
```

6.4.4　嵌套函数与作用域

在 JavaScript 中我们可以对函数进行嵌套声明。也就是说，可以在一个函数中声明另一个函数。这时，可以在内部的函数中访问其外部函数的作用域。在 5.4 节中提到过，从形式上来说，名称的查找是由内向外的。在最后将会查找全局作用域中的名称。

代码清单 6.7 是个具体例子。在代码清单 6.7 中写的是函数声明语句，如果使用的是匿名函数表达式，效果是相同的。

代码清单 6.7　嵌套函数及其作用域

```
function f1() {
    var x = 1;              // 函数 f1 的局部变量

    // 嵌套函数的声明
    function f2() {
        var y = 2;          // 函数 f2 的局部变量
        print(x);           // 对函数 f1 的局部变量进行访问
        print(y);           // 对函数 f2 的局部变量进行访问
    }

    // 嵌套函数的声明
    function f3() {
        print(y);           // 如果不存在全局变量 y，则会发生 ReferenceError
    }

    // 嵌套函数的调用
    f2();
    f3();
}
```

```
// 对代码清单 6.7 的调用
js> f1();
1
2
ReferenceError: y is not defined
```

6.4.5　变量隐藏

在这里使用了隐藏这一比较专业的术语，它指的是，通过作用域较小的变量（或函数），来隐藏作用域较大的同名变量（或函数）。这种情况常常会在无意中发生，从而造成错误。例如，在下面的代码中，全局变量 n 被局部变量 n 所隐藏。

```
js> var n = 1;              //全局变量
```

```
js> function f() {          // 局部变量隐藏了全局变量
    var n = 2;
    print(n);
    }

// 函数调用
js> f();
2
```

这段代码的功能显而易见。乍一看，类似于代码清单 6.3 或图 6.1 那样的函数作用域以及块级作用域所构成的隐藏并不会引发什么问题。不过，当代码变得更为复杂时，问题就不容易发现了，因此仍需多加注意。

6.5 函数是一种对象

函数也是一种对象。从内部结构来看，它继承于 Function 对象。可以像下面这样通过 constructor 属性验证。至于更具有实际意义的内容，将在之后的 6.6.1 节说明。

```
js> function f() {}   // 函数的内容无关紧要，因此留空
js> f.constructorr;
function Function() {
    [native code]
}
```

将匿名函数赋值给某个变量，与将 Function 对象的引用赋值给某个变量，在本质上是相同的，仅仅是表述方式的不同而已。在通常的语境中，函数的概念等价于 Function 对象的引用。因此，函数的声明与 Function 对象的生成也是等价的。

从形式上来看，代码清单 6.8 中例举的 4 种方式差异巨大，不过从整体上来看，这些代码的功能是类似的，都是先生成一个实体（即没有名字的对象），之后将其与引用了它的名称相结合。

代码清单 6.8　从整体上来看，都是生成了实体并赋予一个引用了该实体的名称

```
var obj = {};
var obj = new MyClass();
var obj = function() {};
function obj() {}
```

可以像下面这样，通过对 Function 函数进行构造函数调用，来生成一个 Function 对象。不过这种用法并不多见。

```
// 函数声明（Function 对象的生成）
js> var sum = Function('a', 'b', 'return Number(a) + Number(b);');
// 最后一个参数是函数体。它之前的参数都是函数的形参。

// 函数的调用
js> sum(3, 4);
7
```

由于函数是一种对象，因此自然可以读写 Function 对象的属性。

```
js> function f() {}   // 函数内部的内容对读写操作不会造成影响，因此在此使用了一个空函数
js> f.foo = 'FOO';    // 对 Function 对象 f 的 foo 属性进行赋值

js> print(f.foo);
FOO
```

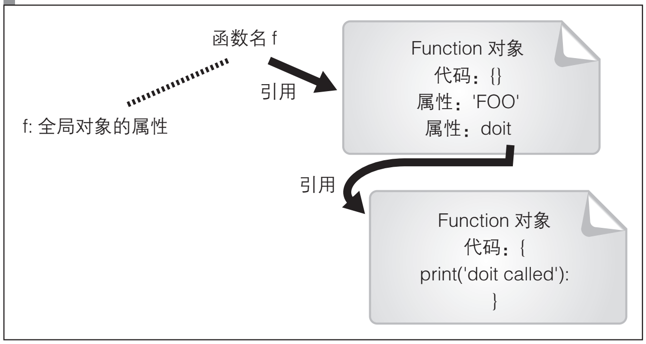
如果对属性赋值以其他的函数，就相当于为该函数（对象）添加了方法。

```
// 接之前的代码
js> f.doit = function() { print('doit called'); }
js> f.doit();
```

```
doit called
```

在这种情况下，一个变量与其所引用的对象之间的关系可能有些复杂，不过只要记住 Function 对象含有一些可执行代码的这一事实，就不会有什么问题（图 6.2）。

图 6.2　变量 f 所引用的 Function 对象（Function 对象中含有引用了其他函数的变量）



函数名与调试的难易度

对象从本质上来说是不具有名称的。由于函数也是一种对象，所以 Function 对象也没有名称。从原则上来说这没有错，不过有必要再对此做一些补充说明。在通过函数声明语句以及匿名函数表达式对函数名进行指定时，在 Function 对象的内部储存了其表示名称。

在下面的代码中，fn_name 的部分将作为 Function 对象的表示名称。

```
function fn_name() { … }              // 函数声明语句
var fn = function fn_name() { … }     // 匿名函数表达式
```

“函数的表示名称”这一说法为本书独创，并非通用术语。我们特意使用这一名称为了与通常的函数名的概念相区分。

函数名是具有 Function 对象的引用的变量名。而“函数的表示名称”则是被储存于 Function 对象内部的名称。虽然无法直接通过“函数的表示名称”来调用函数，不过对于使用函数声明语句的情况，由于在 function 之后所写的名称就是其函数名，所以从表面上来看并没有区别。然而在内部，函数名与函数的表示名称是分别存在的。下面的代码没什么实际意义，不过可以对这一点进行验证。

```
js> function fn() {}
// 函数声明语句。由于函数内部的内容对调用操作不会造成影响，而在此使用了一个空函数
js> var fn2 = fn;      // 这样一来也可以通过函数名 fn2 对其进行调用
js> fn = null;         // 将变量 fn 的值置为 null
js> fn2;               // “函数的表示名”仍然为 fn
function fn() {
}
```

在对 Function 对象进行 print 等操作时会使用函数的表示名称。例如，在显示 constructor 属性所引用的 Function 对象时，所表示的名称即函数的表示名称。而在调试中显示调用栈时，函数的表示名称将发挥更大的作用。

在 JavaScript 程序设计中，与函数声明语句相比，使用匿名函数表达式的情况似乎越来越多。这时，常常会省略写在 function 之后的函数的表示名称。但因为在调试过程中函数的表示名称是非常有用的，所以还请对这一问题多加考虑。

6.6 Function 类

Function 类是用于 Function 对象的类。表 6.2 总结了 Function 类的函数以及构造函数调用。

表 6.2　Function 类的函数以及构造函数调用

函数或是构造函数	说明
Function(p0, p1, …, body)	通过参数 p0,p1,…与函数体 body（字符串）来生成 Function 实例
new Function(p0, p1, …, body)	通过参数 p0,p1,…与函数体 body（字符串）来生成 Function 实例

如果没有特别的理由，我们建议不要通过以上方式，而是使用函数声明语句或匿名函数表达式来生成函数。

表 6.3 总结了 Function 类的属性。

表 6.3　Function 类的属性名

属性名	说明
prototype	用于原型链
length	值为 1

表 6.4 总结了 Function.prototype 对象的属性。

表 6.4　Function.prototype 对象的属性名

属性名	说明
apply(thisArg, argArray)	将 argArray 的所有元素作为参数对函数调用。函数内的 this 引用引用的是 thisArg 对象
bind(thisArg[, arg0, arg1, ...])	返回一个新的 Function 对象。调用此函数时，arg0、arg1 等是实参，函数内的 this 引用引用的是 thisArg 对象
call(thisArg[, arg0, arg1, ...])	将 arg0、arg1 等作为实参对函数调用。函数内的 this 引用引用的是 thisArg 对象
caller	JavaScript 自带的增强功能。表示的是对当前函数调用的函数
constructor	对 Function 类对象的引用
isGenerator()	JavaScript 自带的增强功能。当函数是 generator 时，返回 true（*1）
length	函数的形参的数量
name	JavaScript 自带的增强功能。函数的表示名称（*2）
toSource()	JavaScript 自带的增强功能。求值结果将返回用于函数进行生成的字符串
toString()	将函数体转换为字符串形式并返回

（*1）请参见 7.1.13 节。

（*2）请参见 6.5.1 节。

表 6.5 总结了 Function 类的实例属性。

表 6.5　Function 类的实例属性

属性名	说明
（内部属性）	函数本身的代码
caller	JavaScript 自带的增强功能。表示的是对当前函数进行调用的函数
length	函数的参数的数量
name	JavaScript 自带的增强功能。函数的表示名称（*1）
prototype	用于原型链

（*1）请参见 6.5.1 节。

Function 类的继承

JavaScript 的函数是 Function 对象的对象实例。如果用原型继承相关的术语来解释，即 JavaScript 中的函数的原型对象是 Function.prototype。

可以通过图 6.3 的方式对此进行验证。图 6.3 使用的是函数声明语句，若使用匿名函数也是一样的结果。

图 6.3　对 Function 类的继承的验证

```
js> function fn() {}

js> fn.constructor === Function;            // 构造函数
true
js> fn._proto_ === Function.prototype;      // 原型对象
true
```

下面的说法可能会有些不易于理解，不过 Function 函数也是 Function 类的对象实例。它们之间具有如下自我引用的关系。

```
js> Function === Function.constructor;
true
js> Function._proto_ == Function.prototype;
true
```

函数是原型继承于 Function 对象（Function 类）的。这就意味着，对于一个函数，可以读取访问其在表 6.4 中所列出的属性（或是对其进行方法调用）。对这一关系的说明有些复杂，不过其实际的作用也就仅此而已。

6.7　嵌套函数声明与闭包

6.7.1　对闭包的初步认识

为了让没有接触过闭包（closure）这个词的人也能理解其含义，这里先撇开语言的严密性，试着从表面上说明。请看下面的代码示例。

```
js> var fn = f();       // 将函数 f 的返回值赋值给变量 fn
js> fn();               // 对 fn 的函数调用
1
js> fn();
2
js> fn();
3
```

函数 f 的内容将在之后说明，在此先不必在意。函数 f 的返回值是函数（对象的引用），这里将其赋值给了变量 fn。在调用函数 fn 时，其输出结果每次都会增加 1。可以以 Java 的方式对其内部实现作如下推测：该对象具有一个私有域作为内部计数器。当方法被调用时，内部计数器将会增加。不过从外表上只能看到函数调用这一行为。

函数 f 的内部结构如下所示。更为详细的内容将在下一节说明。

```
function f() {
    var cnt = 0;
    return function() { return ++cnt; }
}
```

从表面上来看，闭包是一种具有状态的函数。如果只是希望简单地使用闭包，这样理解就可以了。或者也可以将闭包的特征理解为，其相关的局部变量在函数调用结束之后将会继续存在。

例如，在上面的例子中，函数内的局部变量 cnt 在函数 f 的调用之后依然有效。

6.7.2　闭包的原理

■ 嵌套的函数声明

闭包的前提条件是需要在函数声明的内部声明另一个函数（即嵌套的函数声明）。下面是一个嵌套函

数声明的简单的例子。在下面的例子中使用的是函数声明语句，若使用匿名函数表达式也是一样的结果。

```
js> function f() {     // 该函数具有嵌套函数声明
        function g() {
          print('g is called');
        }
        g();
     }

// 对函数进行调用
js> f();
g is called
```

在函数 f 中包含函数 g 的声明以及调用语句。在调用函数 f 时，就间接地调用了函数 g。这一行为本身并没有难以理解之处，不过为了能更好地理解该过程，在此对其内部机制进行说明。

在最外层代码中对函数 f 的声明，具有 Function 对象的生成以及通过变量 f 来引用该 Function 对象这两层含义。同时，变量 f 是全局对象的属性。之后将不再使用变量这个词，而改用属性这一术语。

在 JavaScript 中，调用函数时将会隐式地生成 Call 对象。方便起见，我们将调用函数 f 时生成的 Call 对象称作 Call-f 对象。在函数调用完成之后，Call 对象将被销毁。

函数 f 内的函数 g 的声明将会生成一个与函数 g 相对应的 Function 对象。其名称 g 是 Call-f 对象的属性。由于每一次函数调用都会独立生成 Call 对象，因此在调用函数 g 时将会隐式地生成另一个 Call 对象。方便起见，我们将该 Call 对象称作 Call-g 对象。

离开函数 g 之后，Call-g 对象将被自动销毁。类似地，离开函数 f 之后，Call-f 对象也将自动销毁。此时，由于属性 g 将与 Call-g 对象一起被销毁，所以由 g 所引用的 Function 对象将会失去其引用，而最终（通过垃圾回收机制）被销毁。

■ 嵌套函数与作用域

首先我们像下面这样对代码作少许修改。

```
function f() {
    var n = 123;
    function g() {
        print('n is ' + n);
        print('g is called');
    }
    g();
}
```

```
// 对函数进行调用
js> f();
n is 123
g is called
```

这一结果和代码给人的直观感受也可以说是一致的。根据其形式，可以像下面这样来理解其作用域，即在内层进行声明的函数 g 可以访问外层的函数 f 的局部变量（在这里指变量 n）。5.4 节提到过，函数内的变量名的查找是按照先 Call 对象的属性再全局对象的属性这样的顺序进行的。对于嵌套声明的函数，内部的函数将会首先查找被调用时所生成的 Call 对象的属性，之后再查找外层函数的 Call 对象的属性。这一机制被称为作用域链。

■ 嵌套函数的返回

我们进一步对之前的代码作如下修改。

```
function f() {
    var n = 123;
    function g() {
        print('n is ' + n);
        print('g is called');
```

```
    }
    return g;     // 在内部返回已被声明的函数（未对函数进行调用）
}
```

```
// 对函数进行调用
js> f();
function g() {
    print('n is ' + n);
    print('g is called');
}
```

由于 return g 语句，函数 f 将会返回一个 Function 对象（的引用）。调用函数 f 的结果是一个 Function 对象。这时，虽然会生成与函数 f 相对应的 Call 对象（Call-f 对象）（并在离开函数 f 后被销毁），但由于不会调用函数 g，所以此时还不会生成与之对应的 Call 对象（Call-g 对象），请对此加以注意。

■ 闭包

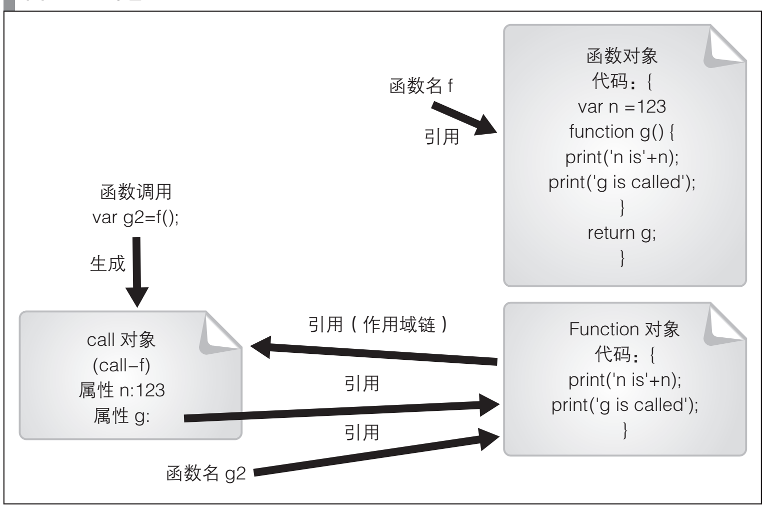
现尝试将函数 f 的返回值赋值给另一个变量。虽然也可以直接调用函数而不赋值，不过为了使整个过程更易于理解，还是采用赋值操作。变量名为 g2，即通过 g2 来调用函数。

```
js> var g2 = f();      // 将返回的函数赋值给变量
js> g2();              // 调用函数（函数 f 内的函数 g）
n is 123
g is called
```

这一结果说明可以从函数 f 的外部调用函数 g。进一步说，这说明函数 f 的局部变量 n 在函数 f 被调用之后依然存在。从表面上看，这与 Java 等其他的过程式程序设计语言中的规则是相反的（一般来说，离开函数之后其局部变量就会无效）。

函数 f 被调用时生成的 Call 对象（Call-f 对象）的属性 g 所引用的 Function 对象（已经反复强调，对象本身是没有名称的），为 g2 所引用。只要引用的变量还存在，对象就不会成为垃圾回收机制的目标。因此，只要名称 g2 还存在，Function 对象就会存在。该 Function 对象具有 Call-f 对象的引用（被用于作用域链）。因此，如果被名称 g2 所引用的这个 Function 对象还存在，Call-f 对象也会继续存在。这就是为什么在离开了函数 f 之后局部变量 n 依然存在的原因。图 6.4 对以上关系进行了整理。

图 6.4　闭包



像下面这样调用函数 f 两次之后，g2 与 g3 将分别引用两个不同 Function 对象。之后，这两个 Function 对象将会引用各自不同的 Call-f 对象，因为 Call 对象会在每次函数调用时被独立生成。

```
js> var g2 = f();
js> var g3 = f();
```

为了更清晰地展示 g2 与 g3 所引用的函数在被调用时的不同，代码清单 6.9 使用了一种比较有技巧性的做法。由于这两个 Call-f 对象是不相同的，因此可以分别通过 g2 与 g3 对各自对象的属性（从函数 f 的角度来看即局部变量 n）访问。

代码清单 6.9　闭包

```
function f(arg) {
    var n = 123 + Number(arg);
    function g() {
        print('n is ' + n);
        print('g is called');
    }
    return g;
}
```

```
// 对代码清单 6.9 的调用
js> var g2 = f(2);
js> var g3 = f(3);

js> g2();
n is 125
g is called

js> g3();
n is 126
g is called

js> var n = 7;              // 对全局变量 n 进行定义，但这对结果没有影响
js> g3();
n is 126
g is called
```

■ **闭包与执行环境**

现不考虑内部执行过程，而是以抽象的方式重新考察对 g2 与 g3 的调用结果不同的这一现象。这意味着可以通过同一段代码生成具有不同状态的函数。这就是所谓的闭包。说得专业一点，闭包指的是一种特殊的函数，这种函数会在被调用时保持当时的变量名查找的执行环境。

由于本书并非理论专著，所以仅将闭包解释为具有状态的函数。这种较为通俗的说明会更易于理解。不过，闭包仅仅是保持了变量名查找的状态，而并没有保持对象所有的状态，对此请加以区分。也就是说，闭包虽然会保持（在嵌套外层进行函数调用时被隐式地生成的）Call 对象，但无法保持 Call 对象的属性所引用的之前的对象的状态。由于这一原因而产生的一些需要注意的地方，将在下一节中进行说明。

像下面这样，在匿名函数表达式中直接使用 return 语句的写法很常见，所以请记住这种闭包的习惯用法。

```
function f(arg) {
    var n = 123 + Number(arg);
    return function () {
        print('n is ' + n);
        print('g is called');
    };
}
```

6.7.3　闭包中需要注意的地方

如果在函数 f 内有两个函数声明，这两者将会引用同一个 Call-f 对象（代码清单 6.10）。这是在使用

JavaScript 的闭包时容易出错的地方。

代码清单 6.10　闭包中需要注意的地方

```
function f(arg) {
    var n = 123 + Number(arg);
    function g() { print('n is ' + n); print('g is called'); }
    n++;
    function gg() { print('n is ' + n); print('gg is called'); }
    return [g, gg];
}
```

```
// 对代码清单 6.10 的调用
js> var g_and_gg = f(1);
js> g_and_gg[0]();              // 对闭包 g 的调用
n is 125
g is called

js> g_and_gg[1]();              // 对闭包 gg 的调用
n is 125
gg is called
```

函数 g 与函数 gg 保持了各自含有局部变量 n 的执行环境。由于声明函数 g 时的 n 值与声明函数 gg 时的 n 值是不同的，因此闭包 g 与闭包 gg 貌似将会表示各自不同的 n 值。但实际上两者将会表示相同的值。这是因为两者引用了同一个 Call 对象（Call-f 对象）。

6.7.4　防范命名空间的污染

■ 模块

接下来，我们来介绍几种运用了闭包的实际示例。

在 JavaScript 中，在最外层代码（函数之外）所写的名称（变量名与函数名）具有全局作用域，即所谓的全局变量与全局函数。如果没有像本书第 6 部分所介绍的 CommonJS 那样另外提供模块功能，JavaScript 的程序代码即使在被分割为多个源文件之后，也能相互访问其全局名称。在 JavaScript 的语言规范中不存在所谓模块的语言功能。

因此，对于客户端 JavaScript，如果在一个 HTML 文件中对多个 JavaScript 文件进行了读取，则它们相互之间的全局名称会发生冲突。也就是说，在某个文件中使用的名称无法同时在另一个文件中使用。即使在独立开发中这也很不方便，在使用他人开发的库之类时就更加麻烦了。

此外，全局变量还降低了代码的可维护性。不过也不能就简单下定论说问题只是由全局变量造成的。这就如同在 Java 这种语言规范不支持全局变量的语言中，同样可以很容易地创建出和全局变量功能类似的变量。也就是说，不应该只是一味地减少全局变量的使用，而应该形成一种尽可能避免使用较广的作用域的意识。对于较广的作用域，其问题在于修改了某处代码之后，会难以确定该修改的影响范围，因此代码的可维护性会变差。

■ 避免使用全局变量

从形式上来看，在 JavaScript 中减少全局变量的数量的方法是很简单的。首先我们按照下面的代码这样预设一下全局函数与全局变量。

```
// 全局函数
function sum(a, b) {
    return Number(a) + Number(b);
}

// 全局变量
var position = { x:2, y:3 };
```

再像下面这样，借助通过对象字面量生成的对象的属性，将名称封入对象的内部。于是从形式上来看，全局变量就减少了。

```
// 封入对象字面量中
var MyModule = {
    sum: function(a, b) {
        return Number(a) + Number(b);
    },
    position: { x:2, y:3 }
};
```

```
// 对其进行调用
js> MyModule.sum(3, 3);
6
js> print(MyModule.position.x);
2
```

上面的例子使用了对象字面量，不过也可以像下面这样不使用对象字面量。

```
var MyModule = {};                    // 也可以通过 new 表达式生成
MyModule.sum = function(a, b) { return Number(a) + Number(b); };
MyModule.position = { x:2, y:3 };
```

在这个例子中，方便起见，我们将 MyModule 称为模块名。如果完全采用这种方式，对于 1 个文件来说，只需要 1 个模块名就能消减全局变量的数量。当然，模块名之间仍然可能产生冲突，不过这一问题在其他的程序设计语言中也是一个无法被避免的问题。

通过这种将名称封入对象之中的方法，可以避免名称冲突的问题。但是这并没有解决全局名称的另一个问题，也就是作用域过广的问题。在上面的代码中，通过 MyModule.position.x 这样一个较长的名称，就可以从代码的任意一处访问该变量。

■ 通过闭包实现信息隐藏

JavaScript 语言并没有提供可用于信息隐藏的语法功能，不过灵活运用闭包之后，就能够使得名称无法从外部被访问。代码清单 6.11 是个具体例子。不过代码清单 6.11 的代码仅仅是为了对此说明，并没有实际意义。

代码清单 6.11　使用了闭包的模块

```
// 在此调用匿名函数
// 由于匿名函数的返回值是一个函数，所以变量 sum 是一个函数
var sum = (function() {
        // 无法从函数外部访问该名称
        // 实际上，这变成了一个私有变量
        // 一般来说，在函数被调用之后该名称就将无法再被访问
        // 不过由于是在被返回的匿名函数中，所以仍可以继续被使用
        var position = { x:2, y:3 };

        // 同样是一个从函数外部无法被访问的私有变量
        // 将其命名为 sum 也可以。不过为了避免混淆，这里采用其他名称
        function sum_internal(a, b) {
            return Number(a) + Number(b);
        }

        // 只不过是为了使用上面的两个名称而随意设计的返回值
        return function(a, b) {
            print('x = ', position.x);
            return sum_internal(a, b); };
        }
        )();
```

```
// 调用代码清单 6.11
js> sum(3, 4);
x = 2
```

```
7
```

代码清单 6.11 可以被抽象为下面这种形式的代码。在利用函数作用域可以封装名称，以及闭包可以使名称在函数调用结束后依然存在这两个特性后，信息隐藏得以实现。

```
(function() { 函数体 })();
```

像上面这样，当场调用匿名函数的代码看起来或许有些奇怪。一般的做法是先在某处声明函数，之后在需要时调用。不过这种做法是 JavaScript 的一种习惯用法，还请加以掌握。

代码清单 6.11 的匿名函数的返回值是一个函数，不过即使返回值不是函数，也同样能采用这一方法。例如，可以像代码清单 6.12 这样返回一个对象字面量以实现信息隐藏的功能。

代码清单 6.12　将代码清单 6.11 的返回值更改为对象字面量

```
var obj = (function() {
        // 从函数外部无法访问该名称
        // 实际上，这是一个私有变量
        var position = { x:2, y:3 };

        // 这同样是一个无法从函数外部进行访问的私有函数
        function sum_internal(a, b) {
            return Number(a) + Number(b);
        }

        // 只不过是为了使用上面的两个名称而随意设计的返回值
        return {
            sum:function(a, b) { return sum_internal(a, b); },
                x:position.x
            };
        }
    )();
```

```
// 调用代码清单 6.12
js> obj.sum(3, 4);
7

js> print(obj.x);
2
```

本节所使用的方法，也能够直接被用于下一节中使用了闭包的类中。

6.7.5　闭包与类

5.7.3 节已经对 JavaScript 的类的定义作了介绍。对于构造函数的类来说，存在以下问题。

- **无法对属性值进行访问控制（private 或 public 等）**

JavaScript 没有与访问控制有关的语法结构。不过只要利用函数作用域与闭包，就可以实现访问控制。按照本节介绍的方法，就能够生成无法变更状态的不可变对象。相关内容可参见 5.12 节。

基本的思路就是利用上一节中提到的模块。在上一节中，模块的函数在被声明之后直接就对其调用，而使用了闭包的类则能够在生成实例时调用。即使如此，这种做法在形式上仍然只是单纯的函数声明。下面是一个通过闭包来对类进行定义的例子（代码清单 6.13），这个类与 5.7.3 节中的代码清单 5.9 的 MyClass 是等价的。

代码清单 6.13　代码清单 5.9 中的 MyClass 的闭包实现版本

```
// 用于实例生成的函数
function myclass(x, y) {
    return { show:function() { print(x, y); } };
}
```

```
// 通过代码清单 6.13 生成实例
js> var obj = myclass(3,2);
js> obj.show();
3 2
```

这里再举一个具体的例子，一个实现了计数器功能的类（代码清单 6.14）。请根据注释来理解这种通过闭包实现的类的结构。

代码清单 6.14　实现了计数器功能的类

```
function counter_class(init) {                    // 初始值可以通过参数设定
    var cnt = init || 0;                          // 设置默认参数的习惯做法（参见 5.5 节）

    // 如有必要，可在此声明私有变量与私有函数

    return {
         // 公有方法
         show:function() { print(cnt); },
         up:function() { cnt++; return this; },  // return this 在使用方法链时很方便
         down:function() { cnt--; return this; }
    };
}
```

```
// 使用代码清单 6.14 的示例
js> var counter1 = counter_class();
js> counter1.show();
0
js> counter1.up();
js> counter1.show();
1

js> var counter2 = counter_class(10);
js> counter2.up().up().up().show();     // 方法链
13
```

专栏

表达式闭包

JavaScript 有一种自带的增强功能，称为支持函数型程序设计的表达式闭包（Expression closure）。

从语法结构上来看，表达式闭包是函数声明表达式的一种省略形式。可以像下面这样省略只有 return 的函数声明表达式中的 return 与 {}。

```
var sum = function(a, b) { return Number(a) + Number(b); }
可以省略为
var sum = function(a, b) Number(a) + Number(b);
```

6.8 回调函数设计模式

6.8.1 回调函数与控制反转

回调函数是程序设计的一种方法。这种方法是指，在传递了可能会进行调用的函数或对象之后，在需要时再分别对其进行调用。由于调用方与被调用方的依赖关系与通常相反，所以也称为控制反转（IoC，Inversion of Control）。

由于历史原因，在 JavaScript 开发中我们常常会用到回调函数这一方法，这是多种因素导致的。第一个原因是在客户端 JavaScript 中基本都是 GUI 程序设计。GUI 程序设计是一种很适合使用所谓事件驱

动的程序设计方式。事件驱动正是一种回调函数设计模式。在本书第 3 部分中我们可以看到，客户端 JavaScript 程序设计是一种基于 DOM 的事件驱动式程序设计。

第二个原因是，源于客户端 JavaScript 无法实现多线程程序设计[①]。而通过将回调函数与异步处理相结合，就能够实现并行处理。由于不支持多线程，所以为了实现并行处理，不得不使用回调函数，这逐渐成为了一种惯例。最后一个原因与 JavaScript 中的函数声明表达式和闭包有关。在阅读了本节之后，你就能明白具体的理由。

虽然由于历史原因，在 JavaScript 中我们经常会使用回调函数，但回调函数并非 JavaScript 所独有的特性。笔者认为，在规模达到一定程度的程序设计中，类似于回调函数的设计模式是很普遍的。从框架程序设计、事件驱动、插件等的架构层面，到观察者模式、模板方法模式等设计模式所在的代码技巧层面，都具有一种类似的结构，即一种设计思想的核心是由一些不变的部分或是抽象代码所组成的，而变化的部分或具体的代码，则集中于其外围并能够被扩充。

6.8.2　JavaScript 与回调函数

回到 JavaScript 的话题。在此，我们将会一边介绍回调函数的实现方法，一边对其执行原理进行说明。在本书第 3 部分之后会有一些实例，所以如果需要较贴近实际的代码，可以参考之后的相关章节。在本节对回调函数机制的说明中，我们还会介绍如何在代码中使用回调函数。在实际的程序设计中，通常开发者只会实现被调用方的代码。不过要是能了解另一方的执行原理的话，应该也是会有所帮助的。

■ 回调函数

代码清单 6.15 是简单的模拟了回调函数的代码。需要预先将 emitter 对象注册（register）为回调函数。当 onOpen 事件发生时，回调函数将被调用[②]。对 emitter 来说，这仅仅是对注册的函数进行了调用，不过根据回调函数的含义，更应该关注使用了 emitter 部分的情况。从这个角度来看，注册过的回调函数与之形成的是一种调用被调用的关系。

代码清单 6.15　单纯的函数型回调函数

```
var emitter = {
    // 为了能够注册多个回调函数而通过数组管理
    callbacks:[],
    // 回调函数的注册方法
    register:function(fn) {
            this.callbacks.push(fn);
        }
    // 事件的触发处理
    onOpen:function() {
            for each (var f in this.callbacks) {
                f();
            }
        }
};
```

```
// 使用代码清单 6.15 的实例

// 回调函数的注册
js> emitter.register(function() { print('event handler1 is called'); });
js> emitter.register(function() { print('event handler2 is called'); });

// 对于事件发生的模拟（对回调函数进行调用）
js> emitter.onOpen();
```

① 在 HTML5 的 Web Workers（请参见本书第 4 部分）中客户端 JavaScript 已经支持多线程了，不过这也是最近才开始的。

② 在事件驱动的程序设计方式中，对调用了回调函数的一方会使用事件分发（emit）或事件触发（fire）这样的术语，而以前缀 on 起始的词常常会被用于调用方。代码清单 6.15 的术语可能会有些不规范，不过这么做是为了避免一下子使用大量新词。

```
event handler1 is called
event handler2 is called
```

■ 回调函数与方法

代码清单 6.15 的回调函数只是单纯的函数而不具有状态。如果回调函数具有状态，就能得到更为广泛的应用。在代码清单 6.16 中，我们把回调方改为了对象，于是代码清单 6.15 中的 emitter 变为了接受方法传递的形式。正如注释中所说，这种做法将不会得到预期的结果。

代码清单 6.16　传递方法的回调函数（不会按预期的方式执行）

```
// 使用了代码清单 6.15 中的 emitter

function MyClass(msg) {
    this.msg = msg;
    this.show = function() { print(this.msg + ' is called'); }
}
```

```
// 使用代码清单 6.16 的实例

// 将方法注册为回调函数
js> var obj1 = new MyClass('listener1');
js> var obj2 = new MyClass('listener2');
js> emitter.register(obj1.show);
js> emitter.register(obj2.show);
```

```
// 对事件发生的模拟（调用回调函数）
js> emitter.onOpen();
undefined is called            // 与期望相背的结果（预期结果为 'listener1 is called'）
undefined is called            // 与期望相背的结果（预期结果为 'listener2 is called'）
```

在代码清单 6.16 中，我们调用回调函数时无法正确显示 this.msg，错误原因在于 JavaScript 的方法内的 this 引用。解决方法有两种，其一是使用 bind，其二是不使用方法而是用对象进行注册。后者在 JavaScript 中并不常用，我们将在之后的专栏中介绍。

下面是使用了 bind 的实现。

```
// 使用了代码清单 6.15 中的 emitter
// 使用了代码清单 6.16 中的 MyClass

// 将方法注册为回调函数
js> var obj1 = new MyClass('listener1');
js> var obj2 = new MyClass('listener2');
js> emitter.register(obj1.show.bind(obj1));
js> emitter.register(obj2.show.bind(obj2));

// 对事件发生的模拟（对回调函数进行调用）
js> emitter.onOpen();
listener1 is called
listener2 is called
```

bind 是 ECMAScript 第 5 版新增的功能，是 Function.prototype 对象的方法（参见表 6.4）。bind 的作用和 5.15 节介绍的 apply 与 call 相同，都是用于明确地指示出方法调用时的 this 引用。对于函数来说，调用了 bind 之后会返回一个新的函数。新的函数会执行与原函数相同的内容，不过其 this 引用是被指定为它的第 1 个参数的那个对象。在调用 apply 与 call 时将会立即调用目标函数，而在调用 bind 时则不会如此，而是会返回一个函数（闭包）。

如果使用了 apply 或 call，就能对 bind 进行独立的实现。事实上，在 ECMAScript 第 5 版推出之前，在 prototype.js 等知名的库中就通过 apply/call 提供了 bind 的自己的实现。有兴趣的读者，可以参见该实现。

■ 闭包与回调函数

最后我们将介绍使用了闭包的回调函数。借助闭包，前面繁复的说明仿佛都不再存在，可以很轻松地实现回调函数，并且还能够像对象一样具有状态。

```
// 使用了代码清单 6.15 中的 emitter

// 将闭包注册为回调函数
js> emitter.register((function() { var msg = 'closure1'; return function() {
print(msg + ' is called'); } })());
js> emitter.register((function() { var msg = 'closure2'; return function() {
print(msg + ' is called'); } })());

// 对事件发生的模拟 ( 调用回调函数 )
js> emitter.onOpen();
closure1 is called
closure2 is called
```

专栏

事件侦听器风格的实现

在此我们介绍一下 Java 等语言中常见的事件侦听器风格的事件方式，以供大家参考。这需要在事件侦听器方对具有特定名称的方法 (以下记为 MyClass 内的 onOpen 方法) 进行实现。有人认为这种事先进行设计的做法很麻烦，不过也有人认为这样一来对象的作用就很明确，所以是一种很好的做法。从历史上来看，JavaScript 几乎没采用过这种编程方式 (通常会使用闭包来实现)。

```
var emitter = {
     callbacks:[],
     register:function(obj) {
               this.callbacks.push(obj);
               },
     onOpen:function() {
              for each (var obj in this.callbacks) {
                  if ('onOpen' in obj) {
                        obj.onOpen();     // 由于调用了方法，this 引用的对象将会与预期结果相同
                  }
              }
     }
};

// 事件侦听器类
function MyClass(msg) {
     this.msg = msg;
     // 必须有 onOpen 方法的实现 ( 与 emitter 内的 'onOpen' in obj 相对应 )
     this.onOpen = function() { print(this.msg + 'is called'); }
}
```

```
// 事件侦听器对象的注册
js> var obj1 = new MyClass('listener1');
js> var obj2 = new MyClass('listener2');
js> emitter.register(obj1);
js> emitter.register(obj2);

// 对事件发生的模拟
js> emitter.onOpen();
listener1 is called
listener2 is called
```

第 7 章 数据处理

本章总结数组、JSON 处理、日期处理以及正则表达式的相关内容。在实际开发中，尤其需要灵活运用数组与正则表达式。接下来，我们将说明 JavaScript 中的数据处理以及相关的习惯用法。

7.1 数组

数组是一种有序元素的集合。在 JavaScript 中，数组的长度是可变的。只要将元素加入数组的尾部，数组的长度就会自动增加。同时，也能够自由改写数组中的每一个元素。其实这并不值得惊讶，反而是理所应当的，因为在 JavaScript 中数组也是一种对象。数组只不过是继承了 JavaScript 的对象的一些性质而已。对于这种性质的意义，之后会再次进行说明。

7.1.1 JavaScript 的数组

在 JavaScript 中，数组可以通过字面量与 new 表达式两种方法生成。通过 new 表达式的生成将在下一节中说明，这里先展示一个数组字面量的例子。

```
// 数组字面量的例子

js> var arr = [3, 4, 5];
js> typeof arr;            // 对数组进行 typeof 运算之后的结果是 object
object
```

数组字面量的书写方式是在中括号（[]）中列出数组元素，并通过逗号相分隔。不含有元素的数组的长度为零。在 JavaScript 中，我们通常会先生成一个长度为零的数组，之后再向其中添加元素。

在 JavaScript 中，我们可以将任意的值或者对象的引用指定为元素，并且不需要确保数组中元素类型的一致性。由于已经知道了可以将任意类型的值赋值给某一变量，因此或许大家不会对这一特性感到惊讶，不过，这确实是与 Java 数列的不同之处。在 Java 中，原则上同一数组中的元素必须类型一致。一方面，JavaScript 的高自由度确实很方便，但另一方面也可能会造成由于意外的数据类型转换而引起的元素赋值错误，所以对此请多加注意。

```
// 不需要确保各个元素的类型一致

js> var s = 'bar';
js> var arr = [1, 'foo', s, true, null, undefined, {x:3, y:4}, [2, 'bar'], function(a, b)
{return Number(a) + Number(b);}];
js> print(arr);
1,foo,bar,true,,,[object Object],2,bar,function (a, b) {
    return Number(a) + Number(b);
}
```

在书写数组字面量时，还可以省略一些中间的元素。被省略元素的值将被认为是 undefined 值。

```
// 中间元素被省略的数组

js> var arr = [3,,5];
```

```
js> print(arr[0], arr[1], arr[2]);
3 undefined 5
```

在 ECMAScript 中，如果像下面这样在书写数组字面量时以逗号作为结尾，则该逗号将会被忽略。不过在旧版本的 Internet Explorer 中，这一会引起错误的原因已经广为人知了。因此，应该避免在数组的最后使用逗号。

```
js> var arr = [3,4,];
js> print(arr.length);     // 在 ECMAScript 标准下，将会忽略最后的逗号
2
```

7.1.2　数组元素的访问

可以通过中括号运算符（[] 运算符）来访问数组的元素，[] 内所写的是下标的数值。下标由 0 开始。如果该下标没有相对应的元素，则会获得 undefined 值。

```
// 使用数组的例子

js> var arr = [3, 4, 5];
js> print(arr[0], arr[1], arr[2], arr[3]);
3 4 5 undefined
```

可以将任何结果为数值的表达式作为下标使用。从内部来看，作为下标的表达式将被作为字符串来求值，然后以数值的方式来使用。因此，像下面这样，把能够被解释为数值的字符串作为下标来使用也没问题。不过，这样一来，代码的可读性会变差，因此并不推荐这种做法。

```
// 接之前的代码

js> var s = '2';
js> print(arr[s]);         // 访问下标为 2 的元素
5

js> print(arr[s + 1]);
undefined                  // 需要注意，这里访问的是 arr[21]（参见 4.31 节）

js> var one = { toString:function() { return '1' } };
// 该对象被转换为字符串型之后结果为 '1'
js> print(arr[one]);
4
```

如果将中括号运算符写在赋值表达式的左侧，则可以改写相应的元素。

```
// 改写数组的元素（接之前的代码）

js> arr[2] = arr[2] * 2;   // 改写下标为 2 的元素的值
10
js> print(arr);
3,4,10
```

如果在赋值表达式左侧所写的下标超过了元素数量，则会向数组增加新的元素。新增的元素下标值不必紧接着现有元素的个数。这时，如果访问中间被跳过的元素，则会返回 undefined 值。

```
// 接之前的代码

js> arr[3] = 20;
// 元素数量为 3 的时候，如果赋值给下标为 3 的元素（第 4 个元素），就会新增一个元素。
20
js> print(arr);
3,4,10,20

js> arr[10] = 100;         // 如果对下标为 10（第 11 个元素）进行赋值，元素的数量就会变为 11
100
js> print(arr);
```

```
3,4,10,20,,,,,,,100
js> print(arr.length);
11
js> print(arr[4]);    // 如果访问被跳过的元素，则会返回 undefined 值
undefined
```

7.1.3 数组的长度

在一个数组之后写上点运算符与 length 就能够获得该数组的长度。数组的长度值为数组中最后一个元素的下标值加 1。之所以用这样稍显复杂的表达方式，是因为如果生成的是元素之间存在间隙的数组，元素的数量与数组的长度不同。请看下面的具体示例。

```
js> var arr = [2,,,,3];         // 元素间有空隙的数组（元素数量为 2）
js> print(arr.length);          // 与元素数量不同，数组的长度为 5
5
```

在向末尾添加了元素之后，length 的值将会自动增加（图 7.1）。如果在添加元素时跳过了一些中间元素，length 的值则是最后的那个元素的下标值减去 1。

图 7.1 数组的 length 值的自动计算

```
js> var arr = ['zero', 'one', 'two'];
js> arr[arr.length] = 'three'; // 借助 arr.length 向数组的末尾添加元素是一种习惯用法
js> print(arr);
zero,one,two,three
js> print(arrr.length);         // 自动增加
4

js> arr[100] = 'x';             // 添加元素时跳过了一些中间元素
js> print(arr.length);          // 自动增加
101
```

还可以显式地更改 length 的值（图 7.2）。在进行改写之后数组的长度也会相应发生改变。如果该值变小，超出部分的元素将被舍去。如果该值变大，新增部分的元素将是 undefined 值。

图 7.2 更改数组的长度

```
js> var arr = ['zero', 'one', 'two'];
js> arr.length = 2;             // 将数组的长度缩短
js> print(arr);                 // 最后一个元素将会丢失
zero,one

js> arr.length = 3;             // 恢复（加长）数组至原来的长度
js> print(arr);                 // 新增的部分是 undefined 值
zero,one,
js> typeof arr[2];
undefined
```

从内部来看，数组长度就是 length 属性，所以也可以像下面这样，通过中括号运算符对其访问。不过，这除了增加代码长度之外没有任何好处，所以一般并不会这样使用。

```
js> print(arr['length']);
```

7.1.4 数组元素的枚举

for 语句是最常用的数组元素的枚举方式。下面是一个例子。

```
// 枚举数组 arr 的所有元素的一种常用方法
for (var i = 0, len = arr.length; i < len; i++) {
    print(arr[i]);
}
```

虽然通过 for in 语句或 for each in 语句也可以枚举数组元素，但它们无法保证枚举的顺序。如果要确保枚举按期望的顺序进行，请使用 for 语句。

除了使用 for 语句这样的循环语句（loop 语句），还有一些可以按顺序调用数组中各个元素的方法。可以通过这样的方法来实现数组元素的枚举。由于其内部机制仍然是一种循环语句，所以也被称为内部循环。

专栏

数组长度的上限

在 ECMAScript 中，JavaScript 的数组长度的上限是 2 的 32 次方。这与 JavaScript 中数值的上限值是不同的，请加以注意。2 的 32 次方以上的数值仅会被识别为属性名而非数值。因此，虽然看似能够使用大于 2 的 32 次方的数字来新增元素，但这并不是数组的元素，所以 length 的值并不会因此而自动增加。如果超过了边界值就可能引发严重的错误，不过元素数量超过了 2 的 32 次方的数组非常少见，所以通常来说无需对这一问题过分在意。

```
js> var arr = [];

js> arr[Math.pow(2, 32) - 2] = '';
js> print(arr.length);                // 数组长度为 2^32-1
4294967295

js> arr[Math.pow(2, 32) - 1] = '';    // 尽管看起来似乎是成功增加了元素……
js> print(arr.length);                // 数组的长度并没有发生变化
4294967295

js> Object.keys(arr);
["4294967294", "4294967295"]          // 2^32-1 以属性的形式存在，而没有被识别为数组元素的下标
```

ECMAScript 第 5 版有多个这种类型的内部循环方法。下面将对其中最具代表性的 forEach 方法进行介绍。forEach 方法的参数应该是一个能够被数组中的各个元素调用的函数（回调函数）。

下面这样的代码能够实现对数组中所有元素的枚举。

```
arr.forEach(function(e) { print(e); })
```

有三个参数被传递给了回调函数，它们分别是元素、下标值以及数组对象。下面是一个具体的例子。

```
js> var arr = ['zero', 'one', 'two'];

// 回调函数的参数
// 参数 e：元素值
// 参数 i：下标值
// 参数 a：数组对象
js> arr.forEach(function(e,i,a) { print(i, e); });
0 zero
1 one
2 two
```

还可以将回调函数内的 this 引用所指向的对象指定为 forEach 的第 2 参数，在此不再赘述。

7.1.5　多维数组

由于任意内容都可以被指定为数组的元素，因此数组本身自然也可以成为另一个数组的元素。某个值若被指定为了数组的元素，可以像下面这样，通过连续使用多个 [] 运算符来访问元素。这样一来就实现了多维数组（细心的读者可能已经发现，数组甚至还可以将其自身作为该数组的元素）。

```
// 数组的元素也是数组（多维数组）

js> var arr_of_arr = [1, ['zero', 'one', 'two', 'three']];
js> print(arr_of_arr[1][1]);
one
```

7.1.6 数组是一种对象

在 JavaScript 中，数组是一种对象。从内部来看，它是 Array 对象（Array 类）的对象实例。因此，也可以通过 new 表达式来调用 Array 的构造函数以生成数组。

根据具体情况，可以以不同的方式来解释传递给 Array 构造函数的参数。如果参数的数量为 1 且是一个数值，它的含义是数组的长度（元素数量）；如果参数的数量大于等于 2，则这些参数代表的是数组的元素。请看图 7.3 的具体例子。

图 7.3 调用 Array 构造函数的例子

```
js> var arr = new Array(5);          // 对于参数只有 1 个的情况，该参数将会成为数组的长度
js> print(arr);
,,,,

js> var arr = new Array(3, 4, 'foo');    // 参数将会成为数组的元素
js> print(arr);
3,4,foo

js> var arr = new Array('5');
// 由于不会发生隐式的数据类型转换而将该参数转换为数值类型，因此这一参数将被认为是数组中下标为 0 的元素

js> print(arr);
5
```

虽然前文介绍了通过 new 表达式生成数组的方式，不过，如果没有特别的理由，最好还是使用字面量表达式来生成数组，因为通过字面量的表达方式更为简单。通过数组字面量表达式生成的数组也是 Array 的实例对象，可以像下面这样对这一点进行确认。

```
js> var arr = [];                    // 通过数组字面量来生成数组对象
js> arr.constructor;                 // 实际上这与通过 new Array() 所生成的对象没有区别
function Array() {
    [native code]
}
```

在通过 new 表达式生成数组时，根据参数数量的不同，参数的含义也会发生改变，而这常常会引起错误。为了避免发生预料之外的错误，我建议不要使用这一方式。不过在一些特定情况下，使用 new 表达式反而更好。例如，在生成数组的同时指定数组长度时，new 表达式更为方便。

举例来说，通过数组字面量来生成一个元素数量为 100 且每个元素的值都未定的数组，虽然也并非无法做到（需要在 [] 内书写 100 个），但会非常麻烦。这时最好使用 new 表达式。此外，由于在新增元素时数组的长度将会自动增加，所以并不一定要在生成数组的同时指定数组的长度。之所以要求同时指定数组的长度，一方面是为了提高执行效率，另一方面是为了使数组的意义更为明确从而提高代码的可读性。

下面是对数组对象的方法进行调用的例子。关于其他一些可以被调用的方法，请参见之后的 7.1.7 节。

```
// 对数组对象的方法进行调用的例子

js> var arr = ['zero', 'one', 'two'];
js> arr.join('_');                   // 对 join 方法进行调用
"zero_one_two"

js> [3, 4, 5].join('_');             // 也可以直接对数组字面量进行方法调用
3_4_5
```

中括号运算符在这里的作用是用于访问数组的元素，其实，这就是在访问对象的属性。也就是说，从内部来看，下标值 0 或 1 之类的数值其实是数组对象的属性名。可以像图 7.4 中那样通过多种手段确认。

在图 7.4 中可以看到，length 也是属性名之一。不过 for 语句并不会对其枚举。这是因为 length 属性的 enumerable 这一属性为假。关于 enumerable 属性，请参见 5.10 节。

图 7.4　数组的属性

```
js> var arr = ['zero', 'one', 'two'];

js> for (var n in arr) { print(n); }    // 下标值的枚举，即属性名的枚举
0
1
2

js> Object.keys(arr);                   // 属性名的枚举
["0", "1", "2"]

js> Object.getOwnPropertyNames(arr);    // 属性名的枚举（忽略 enumerable 属性）
["length", "0", "1", "2"]

js> '0' in arr;                         // 对下标 0 是否存在进行检验
true

js> 0 in arr;                           // 数值 0 将会被转换为字符串型 '0' 以进行检验
true

js> 3 in arr;                           // 对下标 3 是否存在进行检验
false

js> 'length' in arr;                    // 对 length 属性是否存在进行检验
true
```

5.9 节已经说明了在 JavaScript 将对象作为关联数组使用的情况。如果以这种方式来解释的话，JavaScript 中的数组就可以被看作键值恰巧是连续数值的关联数组。此外需要说明的是，如果没有以正整数对一个数组对象进行 [] 运算，该值就会被解释为属性名，而进行属性访问操作。请看下面的例子。

```
js> var arr = ['zero', 'one', 'two'];
js> arr.x = 'X';                        // 向数组对象中添加属性 x
js> for (var p in arr) { print(p); }
0
1
2
x
```

之后的 7.1.8 节将再次说明数组对象所具有的意义。

7.1.7　Array 类

Array 类是一种作为数组对象使用的类。表 7.1 总结了 Array 类的函数以及构造函数调用。

表 7.1　Array 类的函数以及构造函数调用

函数或是构造函数	说明
Array([item0, item1, …])	将参数作为元素以生成数组实例
new Array([item0, item1, …])	将参数作为元素以生成数组实例
Array(len)	生成一个以参数 len 为长度的数组实例
new Array(len)	生成一个以参数 len 为长度的数组实例

表 7.2 总结了 Array 类的属性。可以通过例如 Array.isArray(arg) 的形式使用。

表 7.2 Array 类的属性

属性名	说明
prototype	用于原型链
length	值为 1
isArray(arg)	如果参数 arg 是一个数组实例则返回真

表 7.3 总结了 Array.prototype 对象的属性。

表 7.3 Array.prototype 对象的属性

属性名	说明
constructor	对 Array 类对象的一个引用
concat([item0, item1, ...])	把参数作为元素加入某一数组并生成新的数组。如果参数本身就是一个数组，则将这两个数组连接
every(callbackfn[, thisArg])	依次对数组中的各个元素应用 callbackfn 函数。在 callbackfn 返回 false 之后终止
filter(callbackfn[, thisArg])	依次对数组中的各个元素应用 callbackfn 函数，并返回函数的返回值为 true 的元素所组成的新的数组
forEach(callbackfn[, thisArg])	依次对数组中的各个元素应用 callbackfn 函数
indexOf(searchElement, [fromIndex])	返回第一个与 searchElement 一致的元素的下标。也可以通过第 2 参数来设置检索的起始下标。如果没有找到相符的结果，则返回 –1
join(separator)	在数组的元素之间加入分隔符之后生成相应的字符串值
lastIndexOf(searchElement[, fromIndex])	从后向前检索，返回第一个与 searchElement 一致的元素的下标。也可以通过第 2 参数来设置检索的起始下标。如果没有找到相符的结果，则返回 –1
map(callbackfn[, thisArg])	依次对数组中的各个元素应用 callbackfn 函数，返回元素为函数结果的新的数组
pop()	删除数组中最后一个元素后返回该数组
push([item0, item1, …])	将参数添加至数组的末尾
reduce(callbackfn[, initialValue])	将数组的各个元素与之前的函数调用结果作为参数，依次应用 callbackfn 函数，并返回函数调用的最终结果
reduceRight(callbackfn[, initialValue])	从数组的末尾开始向前执行 reduce 操作
reverse()	将数组中的元素逆序置换
shift()	删除数组中的第一个元素后返回该数组
slice(start, end)	生成一个下标由 start 起至 end 的元素所组成的新的数组
some(callbackfn[, thisArg])	依次对数组中的各个元素应用 callbackfn 函数。如果 callbackfn 的结果为 true，则终止
sort(comparefn)	将数组中的元素排序
splice(start, delCount, [item0, item1, …])	删除下标由 start 开始的 delCount 个元素。如果指定了第 3 个参数，则将该参数插入至前述位置
toLocaleString()	将数列转换为与地区相关的字符串值类型
toSource()	JavaScript 自定义的增强功能。求值结果将返回用于函数进行生成的字符串。
toString()	将数组转换为字符串值类型
unshift([item0, item1, …])	将元素添加至数组的头部

在表 7.4 中对 Array 类的实例属性进行了总结。

表 7.4 Array 类的实例属性

属性名	说明
0 以上的整数值	数组元素
length	数组的长度

7.1.8 数组对象的意义

7.1.6 节已经提到过数组是一种对象，且用于元素访问的下标值是其相应的属性名。

那么，像下面这样，通过对象字面量生成的对象和数组是否是等同的呢？

```
js> var fake_arr = { 0:'zero', 1:'one', 2:'two', length:3 }; // 这与 fake_arr =
['zero', 'one', 'two'] 是否相同?
js> print(fake_arr[1]);             // 在这条语句中，这两者貌似是一样的
one
```

结论就是，上面的这一对象（关联数组）并不是数组。

第一个不同点在于 length 属性的 enumerable 属性。

另一个不同点则是数组的 length 属性的值会自动增加。对于数组来说，在新增加元素时，length 属性将会自动增加。通过 push 方法或 unshift 方法来增加元素，也能使上述的 fake_arr 对象产生相同的结果，但仅通过对元素赋值，则无法实现增加 length 属性的值的效果。

7.1.9　数组的习惯用法

本节介绍与数组有关的多种习惯用法。

■ 排序

可以通过 sort 方法对数组的元素值进行排序。如果不使用参数来调用 sort 方法，则会将其作为字符串进行排序。当以字符串的形式进行排序时，排序是通过对 Unicode 的编码值的大小比较来实现的。大小比较的规则请参见 3.3.3 节。下面是个具体例子。

```
js> var arr = ['one', 'two', 'three', 'four', 'five', 'six'];
js> arr.sort();
["five", "four", "one", "six", "three", "two"]
```

排序之后，数组将改变。之后我会说明这一特性的意义。

如果要以字符串之外的方式对数组进行排序，则需要将比较函数作为参数传递给 sort 方法。如果是一个由数值组成的数组，则可以使用下面的比较函数。使用默认的字符串排序方式对数值进行排序时，虽然对于个位数的数值可以获得预期的结果。但是也会产生 10 比 2 更小这样的结果（请确认 '10'>'2' 的结果）。因此需要注意，不能仅仅因为字符串形式的排序在个位数的情况下能得到正确的结果，而误以为它也适用于所有的数值排序。

```
// 数值组成的数组的排序
js> var arr = [1, 0, 20, 100, 55];
js> arr.sort(function(a,b){ return a -b; });
[0, 1, 20, 55, 100]
```

数组中的每个元素都会在排序时，调用被传递至 sort 方法的参数的比较函数。函数在接受了两个元素的值之后将会返回比较的结果。以 x 与 y 为例，如果 x 比 y 大，则会返回正值。也就是说，如果排序时 x 在 y 之后出现，则会返回一个正值[①]。反之则会返回一个负值。如果值的顺序相同，则会返回 0。

上面的说明可能会不容易理解，总之对于数值的情况，如果返回上面这样的减法运算结果，就能够获得符合的结果了。

sort 方法在调换元素时会对这一数组进行改变。改写目标对象的方法被称为破坏性的方法。在 JavaScript 中，数组含有很多破坏性的方法。下面这些都是破坏性的方法。

- pop、push、reverse、shift、sort、splice、unshift

如果只了解 JavaScript，有可能会想当然地认为数组元素顺序发生改变是理所应当的。然而，不对目标数组进行更改而完成排序，并返回一个新的数组的非破坏性的实现方式也是存在的。通常来说，破坏性的方法很容易引起错误，所以应尽可能避免使用[②]。此外，可以使用在 5.12 节介绍的 freeze 方法来防止数组发生意外改变。

使用 sort 方法对被 freeze 的数组排序的话，会像下面这样引发错误。

```
js> var arr = ['one', 'two', 'three'];
js> Object.freeze(arr);
```

① 准确地说，这是升序排列。通常而言，默认的排序方式都是升序排序。

② 破坏性的方法也有其优点，即效率较高。其内存利用效率确实会更好，因而速度也会较快。

```
js> arr.sort();
TypeError: arr.sort() is read-only
```

■ 通过数组来生成字符串

接下来，我们将介绍一种通过 push 与 join 的组合来生成字符串的常用方法。可以通过数组来实现将字符串一部分一部分地拼接起来，生成一个新的字符串。将每一个部分的字符串全都 push 至数组之后，通过 join 将它们拼接成一个字符串。大家普遍认为，这种方法比通过字符串拼接运算（+ 运算或是 += 运算）更为迅速。

不过，运行速度会随着具体的实现而有所不同，因此也不能盲目认为这种方法的性能更好。如有必要，还是亲自测量速度为好。不过由于现在主流的观点都认为该方法性能更优，所以很多现有的代码都采用了这种方法。因此，即使并不需要使用该方法，也有必要读懂其含义。下面是个具体例子。

```
js> var arr = [];
js> arr.push('<div>');
js> arr.push(Date());
js> arr.push('</div>');
js> arr.join('');
"<div>Sun May 22 2011 14:29:01 GMT+0900 (JST)</div>"
```

join 的参数是在拼接字符串时用于分隔每个部分的字符。在上面的例子中传递了一个空字符，所以实际上没有用到分割字符。如果不传递给 join 参数，则默认的分割字符是逗号字符。

与 join 相对应的逆转换是 String 类的 split 方法。它根据分割字符将字符串分割，之后将每一部分的字符串作为元素加入数组，并将该数组返回。split 的第 1 个参数是用于表示分割字符的字符串值，也可以使用正则表达式。

下面是一个以空格作为分割字符来生成数组的具体例子。

```
js> var str = 'Sun May 22 2011 14:45:04 GMT+0900 (JST)';
js> str.split(' ');         // 通过空格对字符串进行分割
["Sun", "May", "22", "2011", "14:45:04", "GMT+0900", "(JST)"]

js> str.split(/\s/);        // 通过空格（以正则表达式的形式）对字符串进行分割
["Sun", "May", "22", "2011", "14:45:04", "GMT+0900", "(JST)"]
```

■ 数组的复制

下面我们来考虑数组的复制。在很多情况下，与复制及破坏性的方法相关的错误非常常见，想必很多人都曾经遇到过。数组也不例外。

由于在数组的赋值时代入的只是其引用，因此实际上并没有复制数组的元素。仅仅是将某一个变量指向了同一个数组实体而已。因为数组是一种对象，所以这一结果是必然的（参见 5.2 节）。下面是一个具体例子。

```
js> var arr = [3, 5, 4];
js> var arr2 = arr;         // 从变量 arr2 的角度来看，它含有和 arr 相同的元素
js> print(arr2);
3,5,4

js> arr2[0] = 123;          // 通过变量 arr2 来修改数组的元素
js> print(arr);             // 在变量 arr 处也能反映出这一修改
123,5,4
```

如果要复制数组的元素，可以使用 concat 方法或 slice 方法。

下面我将分别为大家介绍使用 concat 方法与 slice 方法的实例（图 7.5、图 7.6）。

图 7.5 通过 concat 方法对数组进行复制

```
js> var arr = [3, 5, 4];
js> var arr2 = [].concat(arr);
```

```
js> print(arr2);                // 从变量 arr2 的角度来看，它含有和 arr 相同的元素
3,5,4

js> arr2[0] = 123;              // 通过变量 arr2 来修改数组的元素
js> print(arr);                 // 在变量 arr 处没有发生变化（因为对元素进行了复制）
3,5,4
```

图 7.6　通过 slice 方法对数组进行复制

```
js> var arr = [3, 5, 4];
js> var arr2 = arr.slice(0, arr.length);
js> print(arr2);                // 从变量 arr2 的角度来看，它含有和 arr 相同的元素
3,5,4

js> arr2[0] = 123;              // 通过变量 arr2 来修改数组的元素
js> print(arr);                 // 在变量 arr 处没有发生变化（因为对元素进行了复制）
3,5,4
```

通常，对于对象或数组实体的复制，有深复制与浅复制两种方式。

深复制是一种完全的复制。如果该对象的属性还引用了其他对象，则那些对象也会一起被复制。

而浅复制则只会复制属性值以及元素值，并不会复制相关的引用对象。通过 concat 以及 slice 进行的复制都是浅复制。可以通过下面的方式确认。

```
js> var arr = [ {x:2} ];        // 该数组的元素是某个对象的引用
js> var arr2 = [].concat(arr); // 通过 concat 复制元素
js> arr2[0].x = 123;            // 修改变量 arr2 处的元素所引用的对象
js> print(arr[0].x);            // 在变量 arr 处也能反映出这一修改（这是一种浅复制）
123
```

如果需要使用深复制，则需要自己实现。不过在实际使用中几乎没有必须使用深复制的情况。

■ 元素的删除

如果要删除数组中的元素，可以使用 delete 运算。不过，通过 delete 删除了元素之后，被删除的地方会留下所谓的空余元素。如果要将进行了元素删除操作后的数列中的空隙消除，可以使用 splice 方法。请看图 7.7 中的示例。

图 7.7　元素的删除（splice 方法）

```
js> var arr = ['zero', 'one', 'xxx', 'two', 'three'];
js> delete arr[2];              // 如果仅仅通过 delete 进行删除操作
js> print(arr);                 // 下标为 2 的位置将会留有空位
zero,one,,two,three

js> arr.splice(2, 1);           // 从下标为 2 的位置起删除 1 个元素
js> print(arr);                 // 前面删除数列的元素后留下的空位被除去了
zero,one,two,three
```

■ 筛选处理

7.1.4 节已经介绍过了 forEach 方法。并不应该去关注对元素进行枚举的这一过程，而应该将数组看作集合了各种成员的单一的对象，将枚举视为对该对象进行的一种操作。

将原来的集合看作输入，而将之后生成的集合看作输出的话，这一操作也可以被看作一种函数。从某种意义上来说，它与函数之间只有表现形式上的差别而已。不过，这种表现形式的差别也是很重要的。如果换一个角度看问题，把这看作一种变换处理，就会发现不仅有能够用于枚举数组元素的 for 循环操作，还有能够用于筛选处理以及流水线处理的相关操作。

对于筛选处理或流水线处理，如果将其分为多级进行会比较方便。可以通过链式语法来实现数组方法的多级处理。下面是一个随意设计的使用示例，并没有提供什么实际的功能。

```
js> var arr = ['zero', 'one', 'two', 'three', 'four'];
```

```
// map：该操作将元素字符串的长度作为新的元素并转换为数组
// filter：该操作将筛选出元素中值为偶数的部分
js> arr.map(function(e) { return e.length; }).filter(function(e) { return e % 2 == 0; });
[4, 4]
```

从表 7.3 中可以看到，有很多方法都可以用于这样的筛选处理。有人将这些方法称为迭代器类方法。如果不想通过循环的方式来实现对数组元素的枚举，还可以考虑一下是否能够使用这种类型的方法。

这种方法的优点之一是代码将变得更为简洁。还有一个优点就是这样的方法能够使开发者对破坏性的方法更为敏感，在使用时更加谨慎。这是如果因为要在筛选处理中使用链式语法，就应该避免使用破坏性的方法。

7.1.10 数组的内部实现

在 JavaScript 以外的很多语言中，数组将会隐式地占用一段连续的内存空间。这种隐式的内部实现，使得高效的内存使用以及高速的元素方法成为可能。然而，在 JavaScript 中，数组的实体是一个对象，所以其通常的实现方式并不是占用一段连续的内存空间。

如果你有其他程序设计语言的开发经验，或许会担心 JavaScript 中数组的执行效率是否比较低。其实 JavaScript 中的数组是否会使用连续的内存空间，取决于具体的实现方式。与其他那些确实是使用了连续内存空间的程序设计语言相比，JavaScript 的数组效率的确会有些让人担心，不过实际上，所有的 JavaScript 实现都为了提高其自身的性能而各自花了不少的功夫。

对比一下代码清单 7.1 与代码清单 7.2，我们就能够对不同实现方式中数组的内部实现进行一定程度的推测。顺便说明一下，这里的代码中的 1e7 表示 10 的 7 次方。指数形式的数值字面量（参见表 3.6）在性能测试类的代码中很有用，记住其使用方式的话会方便很多。

代码清单 7.1　访问大量的数组元素

```
var arr = [];
for (var i = 0; i < 1e7; i++) {
    arr[i] = '';
}
```

代码清单 7.2　代码清单 7.1 的对象形式

```
var arr = {};         // 对象
for (var i = 0; i < 1e7; i++) {
    arr[i] = '';
}
```

根据实现方式的不同，代码清单 7.1 与代码清单 7.2 之间的执行速度会有所差异。这其实是数组是否使用了连续的内存空间的一种体现。然而，如果数组在内部总是使用连续的内存空间，下面的代码就应该会占用多达 GB 量级的连续内存。不过在一般的实现方式中，这样的情况是不会发生的。

```
js> var arr = [];
js> arr[1e9] = '';         // 如果数组确实占用了连续的内存空间的话，应该会消耗大量的内存
```

在流行的 JavaScript 实现中，小型的数组（下标值较小的部分）会占用连续的内存空间，而下标值较大的部分，则一般会以类似于对象的属性的方式进行处理 ①。

此外，在 JavaScript 中，也有人提出需要增加 Int32Array 或 Int8Array 这类自定义增强功能，一并备注于此（请参见下面的链接）。

https://developer.mozilla.org/en/JavaScript_typed_arrays

① 不过说到底，还是要看具体的实现方式。

7.1.11　数组风格的对象

7.1.8 节已经说明过，从使用的角度来看，属性名是数值的对象与数组（无论其内部实现是怎样的）差别很小。在 JavaScript 中，有一种不是数组却具有数组风格的对象。其中较为有名的是可以用于访问函数实参的 arguments 对象。在 DOM 的 API 中，数组风格的对象也有很多。

由于可以通过 for 语句来枚举数组风格的对象中的元素，所以在这方面这种类型的对象与数组的用法相同。一般来说，虽然不能使用 Array 类的方法，但借助 JavaScript 自定义的增强功能（JavaScript1.6），可以对满足以下条件的对象（即数组风格的对象）以类似于类方法调用的形式来使用 Array 类的方法（图 7.8）。

- 对象具有 length 属性
- 对象具有数值属性

图 7.8　对数组风格的对象调用 Array 类的方法（JavaScript 的自定义功能）

```
js> var fake_arr = { 0:'zero', 1:'one', 2:'two', length:3 };  // 数组风格的对象

js> fake_arr.join(',');              // 以一般的方式调用 Array 类的方法将会发生错误
TypeError: fake_arr.join is not a function

js> Array.join(fake_arr, ',');       // 以类方法调用的形式使用 Array 类的方法
"zero,one,two"

js> Array.push(fake_arr, 'three'); // 使用了 push 之后 length 属性也会自动增加
js> print(fake_arr.length);
4
js> Array.join(fake_arr, ',');
"zero,one,two,three"
```

此外还有下面这样的更具有广泛性的解决方案。

```
js> Array.prototype.join.call(fake_arr, ',');
"zero,one,two"
```

7.1.12　迭代器

迭代器（Iterator）这一概念并不是 JavaScript 专有的，在其他的程序设计语言中也有这一概念。简单来说，迭代器是一种专门为循环操作而设计的对象。

不那么严格地说，专门为某种操作而设计出一种功能的方式可以被称作抽象化。而程序设计语言中的迭代器，就是一种对循环操作进行了抽象化而得到的功能。对循环操作进行抽象化之后就能发现，只有继续对下一个对象进行处理的功能是必需的。也可以理解成需要从一个有各种元素的集合中取出下一个所需的元素。

在 JavaScript 中有 Iterator 类这样一个自定义增强功能。可以通过构造函数调用或 Iterator 函数的调用来生成一个对象实例。这时，需要将想要枚举的目标对象传递给它的第 1 参数。

下面是一个具体的例子。之后还将会说明第 2 参数的含义。

```
// 迭代器对象的生成方法

js> var arr = ['zero', 'one', 'two'];
js> var it = new Iterator(arr, true);   // 也可以使用 it = Iterator(arr, true)
```

在迭代器对象中含有一个 next 方法。next 方法能够从（对象）元素的集合中返回下一个所需的元素。这时，根据 Iterator 的构造函数的第 2 参数中的旗标的设置不同，结果也会有所不同。其具体的差别请参见图 7.9。

图 7.9 迭代器的使用示例

```
js> var arr = ['zero', 'one', 'two'];

// 仅返回键（第 2 参数为 true）
js> var it = new Iterator(arr, true);
js> it.next();
"0"
js> it.next();
"1"
js> it.next();
"2"
js> it.next();
uncaught exception: [object StopIteration]

// 同时返回键值对（第 2 参数为 false）
js> var it = new Iterator(arr, false);
js> it.next();
[0, 'zero']
js> it.next();
[1, 'one']
js> it.next();
[2, 'two']
js> it.next();
uncaught exception: [object StopIteration]
```

还可以像图 7.10 这样对 Iterator 对象使用 for in 语句。

图 7.10 对 Iterator 对象使用 for in 语句

```
js> var arr = ['zero', 'one', 'two'];

js> it = new Iterator(arr, true);
js> for (var k in it) { print(k); }
0
1
2

js> var it = new Iterator(arr, false);
js> for (var pair in it) { print(pair); }
0,zero
1,one
2,two
```

对于已经存在的对象或数组来说，使用 Iterator 其实并没有太大的作用。这是因为 for in 语句以及 for each in 语句已经提供了足够的抽象化的循环功能。

那么，什么时候 Iterator 能发挥其作用呢？答案是在使用自定义迭代器时。代码清单 7.3 是一个能够返回阶乘的自定义迭代器的例子。

在代码清单 7.3 中有很多古怪的代码，风格与 JavaScript 不太相像，明显是使用了复杂的代码来实现简单的功能。如果使用生成器（下一节会介绍），就能够用更为简洁的代码来实现同样的功能。在此先尽可能地理解其含义即可。

代码清单 7.3 能够返回阶乘结果的自定义迭代器

```
// 迭代器的目标对象
function Factorial(max) {
    this.max = max;
}

// 自定义迭代器
function FactorialIterator(factorial) {
    this.max = factorial.max;
    this.count = this.current = 1;
```

```
}

// 迭代器的实现
FactorialIterator.prototype.next
  = function() {
    if (this.count > this.max) {
        throw StopIteration;
    } else {
        return this.current *= this.count++;
    }
};

// 将 Factorial 与 FactorialIterator 相关联
// _iterator 属性是一种特殊的属性
Factorial.prototype._iterator_ = function() { return new FactorialIterator(this); }
```

```
// 对代码清单 7.3 的调用
js> var obj = new Factorial(5);
js> for (var n in obj) { print(n); }
1
2
6
24
120
```

7.1.13　生成器

和迭代器一样，生成器（Generator）也是 JavaScript 自定义的增强功能，其作用是帮助执行循环处理。从表面上来看，生成器就像一个普通的函数。生成器与通常的函数的不同之处在于是否在内存进行了 yield 调用。一个函数如果在内部进行了 yield 调用，它就是一个隐式的生成器。此外需要注意的是，在 JavaScript 中，yield 是一个保留字。

下面我将通过与普通函数的比较来对具体说明一下生成器。在代码清单 7.4 中，代码的输出结果是由 1 至通过参数传入的最大值的所有数的阶乘。for 循环内的 print 函数用于输出阶乘的计算结果。

代码清单 7.4　输出（print）阶乘的函数

```
function factorial_printer(max) {
    var cur = 1;
    for (var n = 1; n <= max; n++) {
        cur *= n;
        print('cur = '+ cur);
    }
}
```

```
// 对代码清单 7.4 的调用
js> factorial_printer(5);
cur = 1
cur = 2
cur = 6
cur = 24
cur = 120
```

在代码清单 7.4 中的函数 factorial_printer 内的 print 之前调用 yield 函数的话，就变成了代码清单 7.5 中的 factorial_generator 函数。生成器与它的区别仅仅在于是否使用了 yield，其他方面看起来和普通的函数一样。不过如果像普通的函数那样来调用 factorial_generator，不会有任何输出。先不论这时究竟进行了什么操作，至少 print 确实是没有被执行。

调用函数 factorial_generator 之后将会返回一个对象。这个对象称为迭代生成器。调用迭代生成器的 next 方法的话，就可以执行 1 次生成器中的循环。

代码清单 7.5 生成器的例子

```
function factorial_generator(max) {
    var cur = 1;
    for (var n = 1; n <= max; n++) {
        cur *= n;
        yield(cur);
        print('cur = ' + cur);
    }
}
```

```
// 对代码清单 7.5 的调用
// 在调用之后不会有任何可见反应（仅仅返回了一个迭代生成器）
js> factorial_generator(5);
({})

// 在调用了迭代生成器的 next 方法后，将会执行 1 次 factorial_generator 中的循环
js> var g = factorial_generator(5);
js> print(g.next());
1
js> print(g.next());
cur = 1
2
js> print(g.next());
cur = 2
6
js> print(g.next());
cur = 24
120
js> print(g.next());
cur = 120
uncaught exception: [object StopIteration]
```

由于从内部来看迭代生成器是一个迭代器，所以可以将 next 方法调用隐藏至 for in 循环的内部。可以像下面这样输出所有的阶乘结果。

```
// 通过 for in 语句对迭代生成器进行调用
js> var g = factorial_generator(5);
js> for (var n in g) { print('n = ' + n); }
n = 1
cur = 1
n = 2
cur = 2
n = 6
cur = 6
n = 24
cur = 24
n = 120
cur = 120
```

我们可以将生成器直观地理解为一种由于 yield 而处于停止状态的函数。可以在其外部通过 next 方法使循环过程继续进行。在生成器中，常常会使用像代码清单 7.5 中那样的循环语句，不过事实上生成器中并不一定非要包含循环处理。

由代码清单 7.5 可知，在调用 next 函数时，生成器中的循环将会执行一次。更准确地说，这一执行过程与循环没有关系，只是执行生成器中的代码直至下一次调用 yield 处。在代码清单 7.5 中，print 被写在了 yield 之后，但即使将 print 写在 yield 之前，第一次 g.next() 的调用结果也将是输出 1。也就是说，在调用生成器时不会执行生成器内部的任何内容。只有在调用了 next 方法之后才会执行生成器内的代码，直至 yield 处停止。

7.1.14　数组的内包

数组的内包是一种在通过生成器生成数组时的功能。这也是 JavaScript 自定义的增强功能。

可以使用代码清单 7.5 中的 factorial_generator，生成下面这样一个数组。不过需要删去 factorial_generator 内的 print(cur); 这一行。

```
// 使用代码清单 7.5 中的 factorial_generator
js> var factorial_arr = [i for each (i in factorial_generator(10))];
js> print(factorial_arr);
1,2,6,24,120,720,5040,40320,362880,3628800
```

还可以像下面这样通过运算或 if 语句来实现筛选功能。

```
js> var factorial_arr = [i+1 for each (i in factorial_generator(10))];
js> print(factorial_arr);
[2, 3, 7, 25, 121, 721, 5041, 40321, 362881, 2628801]

js> var factorial_arr = [i for each (i in factorial_generator(10)) if (i > 100)];
js> print(factorial_arr);
[120, 720, 5040, 40320, 362880, 3628800]
```

7.2　JSON

JSON 是 JavaScript Object Notation 的缩写，是一种基于 JavaScript 的字面量表达方式的数据格式类型。其标准为 RFC4627。在 ECMAScript 第 5 版的标准中也包含了 JSON 这一类型。

JSON 能够通过 4 种基本数据类型以及 2 种结构化数据类型来表示。

4 种基本数据类型是指字符串值型、数值型、布尔型以及 null 型。结构化数据类型是指对象与数组这两种。

只要将这里的对象理解为 JavaScript 中的对象即可。也就是说，这是一种元素为名称与值的配对的集合。而数组则是以某种特定顺序排列的元素的集合。

表 7.5 总结了 JSON 的标准。实际上这是 JavaScript 中字面量表达方式的一个子集。关于其中的一些不同之处，请参见表 7.5 中的注意点一栏。

表 7.5　JSON 的标准

数据类型	书写示例	注意点
字符串值	"foobar"	不能使用单引号。字符串的默认编码为 UTF-8
数值	123.4	只支持 10 进制书写方式
布尔值	true 或是 false	
null 值	null	
对象	{ "x":1, "y":"foo" }	属性名只能使用字符串的方式表示而不能使用 {x:1} 这样的字面量形式
数组	{ 1, 2, "foo" }	数组中的元素可以被指定为任意类型的值

7.2.1　JSON 字符串

在实际的程序开发过程中，很多操作都包含了 JSON 格式数据类型的字符串（以下简称 JSON 字符串）与 JavaScript 对象间的相互转换。例如，在将 JSON 数据发送至外部时，需要将内部的对象转换为 JSON 字符串之后再传输。而在接收 JSON 数据的场合，先将 JSON 字符串转换为 JavaScript 对象之后，才能不借助专门的 API 对其值进行读取操作。

■ JSON 字符串的分析

在本节中所介绍的原生 JSON 出现之前，可以通过一种名为 eval 函数的方式将 JSON 字符串转换为 JavaScript 对象。传递给 eval 函数的字符串将被看作是 JavaScript 代码并被执行（以进行求值）。因此，如果将属于 JavaScript 对象的字面量形式的一个子集的 JSON 字符串传递给该函数，就会返回一个对象。不过这种方式存在一些问题。

由于被传递的字符串会被作为代码进行求值，所以其中的语句或函数调用也会一起执行。这在接收不被信任的外部 JSON 数据时是非常危险的。此外，eval 函数本身也有一些小问题。例如，eval('{"x":1}') 这样的代码将会引发错误。

因为 eval 函数会将参数解释为 JavaScript 语句，所以 {"x":1} 不会被看作是对象字面量而会被解释为一条在代码块中有一个标签 x 的语句。为了让这条语句能够被解释为对象字面量，必须像 eval('({"x":1})') 一样再使用一组圆括号。

为了解决这类问题，有必要将字符串解释为 JSON 字符串而不是 JavaScript 语句。在这样的背景下，用于分析 JSON 字符串的库出现了。其中有代表性的一种实现是 json2.js[①]。

随着 JSON 的广泛应用，JSON 分析器不再以外部库的形式存在，而是在 JavaScript 的实现中提供了用于分析 JSON 字符串的 API。在 ECMAScript 第 5 版中，在现有实现方式的基础上，原生 JSON 进一步被定义为一种 API。如果使用了原生 JSON 的 API，不必再依靠 eval 函数这样的对代码进行求值的方式，就能对纯 JSON 字符串进行分析。

7.2.2 JSON 对象

JSON 对象是一种用于原生 JSON 分析的对象，无法对其进行构造函数调用。如果用 Java 中的术语来说，它相当于能够直接使用类方法的工具类。

表 7.6 总结了 JSON 对象的属性。图 7.11 是一个 JSON 对象的使用示例。

表 7.6 JSON 对象的属性

属性名	说明
parse(text[, reviver])	对参数 text 这一 JSON 字符串进行分析之后返回一个 JavaScript 对象。reviver 将会对每个属性调用回调函数，并将返回值赋为属性值
stringify(value[, replacer[, space]])	将参数 value 转换为 JSON 字符串。replacer 将会对每个属性调用回调函数，并将返回值赋为属性值。space 则是输出时的一个缩进字符串

图 7.11 JSON 对象的使用示例

```
// 将 JSON 字符串转换为对象
js> var s = '{"x":1, "y":2, "val":"foobar"}';    // JSON 字符串
js> var obj = JSON.parse(s);
js> print(obj.x);
1

// 将对象转换为 JSON 字符串
js> JSON.stringify({x:1, y:2, val:'foobar'});
"{\"x\":1,\"y\":2,\"val\":\"foobar\"}"

// 将 JSON 字符串的数组转换为对象的数组
js> var arr = JSON.parse("[4, 3, 5]");
js> print(arr);
4,3,5
js> Array.isArray(arr);
true

// 将字符串型的 JSON 字符串转换为字符串值
```

① http://www.JSON.org/json2.js

```
js> var s = JSON.parse('"for"');
js> typeof s;
"string"

// 将数值型的 JSON 字符串转换为数值
js> var n = JSON.parse(3);
js> typeof n;
"number"
```

如果对一个格式错误的 JSON 字符串进行分析，则会发生下面这样的错误。所以在使用单引号的字符串以及对象属性名时应加以注意。

```
js> var s = JSON.parse("'foo'");        // 使用单引号的字符串是一种错误的形式
SyntaxError: JSON.parse

js> var arr = JSON.parse("{x:1}");      // 当属性名不是字符串时会发生错误
SyntaxError: JSON.parse
```

7.3 日期处理

Date 类是一种用于日期处理的类。下面是一个简单的使用示例。

```
js> var dt = new Date(); // 如果使用不含参数的构造函数调用，则会生成一个包含了当前时刻的 Date 实例
js> print(dt);
Sat May 07 2011 03:15:21 GMT+0900 (JST)
```

从内部来看，这一时刻是从基准时起经过的毫秒数的整数形式。这里的基准时是指 GMT 标准的 1970 年 1 月 1 日 0 时 0 分。由于这一基准时也称为 epoch，所以从基准时起所经过的时间，有时也被称为 epoch 毫秒或 epoch 值。在 JavaScript 中，正负整数是用 53 位比特来表示的，所以能够支持基准时之前以及之后的足够长的时间（285616 年）。其中在基准时之前的时间通过负的 epoch 毫秒来表示。

下面进一步说明日期处理的细节。日期可以在 4 种形式之间任意相互转换，如表 7.7 所示。

表 7.7　日期数据的表示形式

名称	主要用途
epoch 值	保存于数据库中的值。各种转换的根底。用于计算经过的时间
Date 类	JavaScript 代码的内部形式。用于月份的处理、星期的处理，或星期几的判断
字符串	用于向用户显示日期（包括农历等），或作为用户输入值的形式，以及网络传输时的形式
年月日等数值	用于向用户显示日期，或作为用户输入值的形式

epoch 值是一种纯数值。从某种意义上来说比较容易处理，但数值本身的适用范围也会有所限制。能够称得上具有实际意义的运算，大概也就只有通过减法来求经过的时间而已了。如果为了求 1 个月之后的值，一般不会通过对 epoch 值进行加法计算来实现。其中的理由很容易理解。到底是应该加上 30 天所经过的毫秒数还是应该加上 31 天所经过的毫秒数，又或者这一年是否是闰年，2 月份到底有多少天，诸如此类的问题都通过 epoch 值来计算的话，是不切实际的。

Data 类的作用就是将这些日期处理中的复杂情况隐藏起来，而且 epoch 值与 Date 对象之间的相互转换也很容易。

字符串与数值的表示方法其实是用于输入输出的形式。在实际开发中，应尽可能在输入或输出操作附近将其转换为 Date 类或 epoch 值。这是因为字符串以及数值形式是与地区和时区设置相关的，所以说少使用这两种形式为好。

epoch 值是一个与地区和时区都无关的数值。又因为 Date 对象是与 epoch 值一一对应的，所以同样也是与地区和时区无关的。正因如此，将 Date 对象转换为字符串或数值时，系统当前所处的地区和时区会对结果产生影响。这一过程中包含了很多日期处理中的难点，比如如何判断某一天是星期几（这与时

区相关），如何表示日期（到底是 2011/1/1 还是 2011 年 1 月 1 日之类的问题与地区相关），如何判断是否是节假日（这取决于各个不同的地区）等。

在实际的客户端侧 JavaScript 中，通常不会进行复杂的日期处理。如果在客户端侧处理这类问题，则意味着结果是基于客户端侧的地区和时区得到的。虽然这种做法也有其合理之处，但通常还是会造成数据混乱。

Date 类

表 7.8 总结了 Date 类的函数以及构造函数调用。

表 7.8 Date 类的函数以及构造函数调用

函数或是构造函数	说明
Date()	返回当前时刻的字符串
new Date([year[, month[, date[, hours[, minutes[, seconds[, ms]]]]]]])	返回参数所指定的时刻的 Date 实例
new Date(value)	将参数作为 epoch 值并返回相应的 Date 实例
new Date()	返回当前时刻的 Date 实例

需要注意的是，和其他一些程序设计语言一样，在 JavaScript 中，month 也是由 0 开始计数的。也就是说，一个显示为 2012 年 1 月 1 日的 Date 对象应该以下面这样的方式生成。

```
js> var dt = new Date(2012,0,1);   // 2012 年 1 月 1 日
js> print(dt);
Sun Jan 01 2012 00:00:00 GMT+0900 (JST)
```

表 7.9 总结了 Date 类的属性。可以通过 Date.now() 的形式使用这些属性。

表 7.9 Date 类的属性

属性名	说明
prototype	用于原型链
length	值为 7
now()	返回当前时刻的 epoch 毫秒
parse(string)	对参数中的字符串进行分析并返回相应的 epoch 毫秒
UTC(year, month[, date[, hours[, minutes[, seconds[, ms]]]]])	返回参数所指定时刻的 epoch 毫秒

表 7.10 总结了 Date.prototype 对象的属性。

表 7.10 Date.prototype 对象的属性

属性名	说明
constructor	指向 Date 类对象的一个引用
getDate()	返回日期的数值。日期从 1 开始计。基于时间
getDay()	返回星期几的数值。一个星期从星期日开始计。星期日是 0，星期六是 6。基于本地时间
getFullYear()	返回年的数值。基于本地时间
getHours()	返回小时的数值。小时从 0 开始计。基于本地时间
getMilliseconds()	返回毫秒的数值。毫秒数从 0 开始计。基于本地时间
getMinutes()	返回分的数值。分从 0 开始计。基于本地时间
getMonth()	返回月份的数值。月份从 0 开始计。1 月是 0，12 月是 11。基于本地时间
getSeconds()	返回秒的数值。秒从 0 开始计。基于本地时间
getTime()	返回一个数值形式的时间。即取得当前的 epoch 毫秒值
getTimezoneOffset()	返回时区的偏差量。单位是分钟
getUTCDate()	返回日期的数值。日期从 1 开始计。基于 UTC 时间
getUTCDay()	返回星期几的数值。一个星期从星期日开始计。星期日是 0，星期六是 6。基于 UTC 时间
getUTCFullYear()	返回年的数值。基于 UTC 时间

（续）

属性名	说明
getUTCHours()	返回小时的数值。小时从 0 开始计。基于 UTC 时间
getUTCMinutes()	返回分的数值。分从 0 开始计。基于 UTC 时间
getUTCMonth()	返回月份的数值。月份从 0 开始计。1 月是 0，12 月是 11。基于 UTC 时间
getUTCSeconds()	返回秒的数值。秒从 0 开始计。基于 UTC 时间
getUTCMilliseconds()	返回毫秒的数值。秒从 0 开始计。基于 UTC 时间
setDate(date)	将日期设定为参数指定的值（1–31）。基于本地时间
setFullYear(year[, month[, date]])	将年份设定为参数指定的值。基于本地时间
setHours(hour[, min[, sec[, ms]]])	将小时设定为参数指定的值。基于本地时间
setMilliseconds(ms)	将年毫秒设定为参数指定的值。基于本地时间
setMinutes(min[, sec[, ms]])	将分钟设定为参数指定的值。基于本地时间
setMonth(month[, date])	将月份设定为参数指定的值（0–11）。基于本地时间
setSeconds(sec[, ms])	将秒设定为参数指定的值。基于本地时间
setTime(time)	将 epoch 毫秒设定为参数指定的值。基于本地时间
setUTCDate(date)	将日期设定为参数指定的值（1–31）。基于 UTC 时间
setUTCFullYear(year[, month[, date]])	将年份设定为参数指定的值。基于 UTC 时间
setUTCHours(hour[, min[, sec[, ms]]])	将小时设定为参数指定的值。基于 UTC 时间
setUTCMilliseconds(ms)	将年毫秒设定为参数指定的值。基于 UTC 时间
setUTCMinutes(min[, sec[, ms]])	将分钟设定为参数指定的值。基于 UTC 时间
setUTCMonth(month[, date])	将月份设定为参数指定的值（0–11）。基于 UTC 时间
setUTCSeconds(sec[, ms])	将秒设定为参数指定的值。基于 UTC 时间
toDateString()	将 Date 实例的日期转换为字符串值。基于本地时间
toJSON(key)	将 Date 实例转换为 JSON 格式的字符串值
toISOString()	将 Date 实例转换为 ISO8601 格式的字符串值
toLocaleDateString()	将 Date 实例的日期转换为与地区相关的字符串值。基于本地时间
toLocaleFormat(format)	JavaScript 自定义的增强功能。以 format 字符串所指定的格式将日期转换为字符串。基于本地时间
toLocaleString()	将 Date 实例转换为地区相关的字符串。基于本地时间
toLocaleTimeString()	将 Date 实例所表示的时刻转换为地区相关的字符串。基于地区相关时刻
toSource()	JavaScript 自定义的增强功能。返回用于生成 Date 实例的字符串（即源代码）
toString()	将 Date 实例转换为字符串值。基于本地时间
toTimeString()	将 Date 实例的时刻转换为字符串值。基于本地时间
toUTCString()	将 Date 实例转换为字符串值。基于 UTC 时间
valueOf()	将 Date 实例转换为数值。即取得当前的 epoch 毫秒值

表 7.11 总结了 Date 类的实例属性。

表 7.11　Date 类的实例属性

属性名	说明
[内部值]	日期值
Date 类	JavaScript 代码的内部形式。用于月份的处理、星期的处理以及对星期几的判断

7.4　正则表达式

7.4.1　正则表达式的定义

正则表达式是一种适用于字符串的模式匹配的语言。正则表达式的英语为 regular expression，常常省略为 regex（其发音为 ['redʒeks]）。在 JavaScript 中，我们可以通过正则表达式对象来使用正则表达式。

正则表达式也是一种语言。不过它不是 JavaScript 这样的多功能程序设计语言，而是一种专门为特定用途而设计的语言。这类专为特定用途设计的语言有时称为 DSL（Domain Specific Language，领域专用语言）。正则表达式是程序设计历史上最为成功的一种 DSL，其应用领域主要集中于搜索（search）与替换（replace）。

字符串模式匹配的一个应用实例，就是在字符串中寻找某个特定的字符串，比如在 "you love JavaScript" 这一字符串中找到 "JavaScript" 这一单词。

上面的例子比较简单，所以无需使用正则表达式，直接用字符串类中的方法会更加方便。那么换一种情况，如果要在 "you love JavaScript" 这一字符串中寻找从 l 开始至 e 结束的单词，应该怎样实现呢？可以想象，代码就会因此变得很复杂了。

下面以此为例，说明一下正则表达式的使用方法。首先是“从 l 开始至 e 结束的单词”的描述方法。在正则表达式中，这可以用 \bl\w*e\b 这样一个乍看不知所云的方式来表达。\bl\w*e\b 这一根据正则表达式规则写出的字符串称为模式，其具体的含义将在之后说明。

正则表达式引擎的功能是，在收到了模式之后从目标字符串中寻找该模式。引擎的内部功能是基于从字符串头部开始逐一对字符串进行检索比对的方式扩展而成的。为了提高执行效率，实际的引擎采取了很多改进手段，不过，这些正则表达式引擎的内部原理对于一般的开发者来说是无需了解的。事实上，正则表达式引擎在 JavaScript 的具体实现中也是被隐藏起来的。

■ 正则表达式的写法

对于模式匹配问题来说，如果要通过正则表达式解决，则需要设计出用于描述该模式的语言写法，并在之后将目标字符串与模式传递给正则表达式引擎以供查找。

从字符串的头部开始查找的方法，说到底是一种针对问题设计出算法以写出相应的子程序的方式。为了让这段子程序能够适应多种情况，不得不向它提供多种外部参数。这些参数可能是多个旗标变量，也可能是供某些方法使用的参数，又或者是很多的回调函数。而正则表达式则采用了另一种不同的方式来解决问题。正则表达式的基本思想是，不使用那些需要用到很多参数的子程序，而是设计出一种可以描述问题的语言。引擎作为这种语言的解释器，而实际解决问题的子程序则被隐藏在了引擎内部。引擎的设计或许会相当复杂，不过只要成功设计出来，解决问题的所有难点就都转移到了语言的书写上。这种思考方式也是一种很高级的程序设计技巧，最好记在脑中，或许什么时候就能发挥作用。

7.4.2 正则表达式相关的术语

接下来对正则表达式的术语进行整理。

- 模式
- 输入字符串
- 匹配

查找规则被称为模式（pattern）。用于查找模式的对象字符串被称为输入字符串。在前一节的例子中，“从 l 开始至 e 结束的单词”是一个模式，而 "you love JavaScript" 则是输入字符串。

在输入字符串中寻找与模式相一致的字符串的过程称为匹配（match）。至于是只检测到第一个匹配位置，还是会检测所有的匹配，则因 API 而异。有时，也会将查找到的字符串，或是找到该字符串的位置称为匹配。

7.4.3 正则表达式的语法

正则表达式也是一种语言，因此有其自身的语法。不过，它的语法与 JavaScript 语言的语法的概念有些不同。事实上，从语言的角度来看，将正则表达式看作一种专为模式匹配而设计的表达方式或许理解起来会更加容易[①]。

① 如果要详细说明正则表达式，恐怕需要一整本书的篇幅了。所以本书仅介绍其中的一部分内容。

■ 模式字符串的组成元素

在正则表达式中，模式字符串的组成元素可以分为元字符与字面量字符两种类型。将会按照写在模式中的内容，原封不动地解释正则表达式中的字面量字符。例如，在正则表达式中 "book" 这一字符串是一个模式字符串，这个模式可以与以下输入字符串相匹配。

- "book"
- "books"
- "buy a book"
- "notebook"

正则表达式引擎并不会将 "book" 理解为一个单词，所以自然也会与 "notebook" 等字符串匹配，对此请多加注意。对于仅有字面量字符的情况，正则表达式或许看起来与 String 类的 indexOf 方法没有什么差别。如果要充分利用正则表达式，则需要对元字符加以利用。

表 7.12 列举了一些在 ECMAScript 第 5 版中定义的正则表达式元字符。

表 7.12　JavaScript 正则表达式的元字符（ECMAScript 第 5 版）

元字符	说明
.	任意 1 个字符
\s	空白字符
\S	非空白字符
\w	可以构成单词的字符①
\W	不能构成单词的字符
\d	数字
\D	非数字
\b	单词的边界
\B	不是单词的边界
^	行首
$	行末
X?	字符 X 重复出现 0 次或 1 次
X??	字符 X 重复出现 0 次或 1 次（非贪心法）②
X*	字符 X 重复出现 0 次或更多次
X*?	字符 X 重复出现 0 次或更多次（非贪心法）
X+	字符 X 重复出现 1 次或更多次
X+?	字符 X 重复出现 1 次或更多次（非贪心法）
X{n}	字符 X 重复出现 n 次
X{n}?	字符 X 重复出现 n 次（非贪心法）
X{n,}	字符 X 重复出现 n 次或更多次
X{n,}?	字符 X 重复出现 n 次或更多次（非贪心法）
X{n,m}	字符 X 重复出现至少 n 次至多 m 次
X{n,m}?	字符 X 重复出现至少 n 次之多 m 次（非贪心法）
X|Y	X 或者 Y
[XYZ]	1 个是 X 或者 Y 或者 Z 的字符
[^XYZ]	1 个除了 X、Y、Z 以外的任意字符
(X)	分组（以供之后引用）
\ 数字	对分组的引用③
(?:X)	仅分组④
X(?=Y)	匹配 X 之后接着 Y 的情况
X(?!Y)	匹配 X 之后不接着 Y 的情况

① 包括字母、数字、下划线以及汉字）（原书中并没有提到汉字，而实际上汉字也是被包含在内的。——译者注

② 即尽可能寻找出现次数较少的情况。——译者注

③ 这里的数字是分组出现的序号。——译者注

④ 即不记录分组序号，也不捕获该匹配。——译者注

如果不希望让元字符中存在的字符被解释为元字符，则需要通过反斜杠字符对其进行转义。例如，如果不希望让模式对点字符（.）进行匹配，则要将其写为 \. 的形式。

如果在模式中要匹配反斜杠字符，则需要将其写成 \\。还有一些转义字符的使用方式与 JavaScript 中的转义字符类似（表 7.13）。

表 7.13 JavaScript 正则表达式中的转义字符

特殊字符（转义字符）	意义
\n	换行（LF）
\t	制表
\r	回车换行（CR）
\f	换页
\v	垂直制表
\cX	控制字符。例如 \cA 为 0x01，\cB 为 0x02 等
\xXX	Latin-1 的编码值（X 是 0 到 9 的数字或 a 到 f 的字母）
\uXXXX	Unicode 的编码值（X 是 0 到 9 的数字或 a 到 f 的字母）

表 7.14 总结了正则表达式中可用的旗标。旗标的设定方法将在下一节中说明。

表 7.14 JavaScript 正则表达式中的旗标

旗标	说明
g	全局匹配模式。之后将会详述
i	忽略英文字母的大小写的模式
m	多行模式。^ 与 $ 将会对多行的行首与行末进行匹配

7.4.4 JavaScript 中的正则表达式

在 JavaScript 中，我们可以通过正则表达式对象来使用正则表达式。正则表达式对象是 RegExp 类的对象实例。下面是一个简单的使用示例。

```
// RegExp 的使用示例
js> var reg = new RegExp('^[0-9]'); // 生成一个正则表达式模式为 ^[0-9] 的 RegExp 实例
js> reg.test('foo');               // 对输入字符串 'foo' 进行匹配
false                              // 结果为假
js> reg.test('123');               // 对输入字符串 '123' 进行匹配
true                               // 结果为真
```

还可以通过字面量方式生成一个 RegExp 实例。可以像下面这样在两个 /（斜杠字符）之间书写正则表达式的模式。

```
// 正则表达式字面量
js> var reg = /^[0-9]/;            // 与 new RegExp('^[0-9]') 等价
js> reg.constructor;               // 确认
function RegExp() {
    [native code]
}
```

在使用正则表达式的旗标时，如果是构造函数，将其传递至第 2 参数，如果是字面量，则写在 2 个斜杠字符之后。下面是个具体例子。

```
// 设置全局匹配的旗标
var reg = /^[0-9]/g;
new RegExp('^[0-9]', 'g');

// 设置多个旗标
var reg = /^[0-9]/gi;
new RegExp('^[0-9]', 'gi');
```

虽然对于正则表达式对象的生成来说，new 表达式与字面量表达式在实际操作上没有区别，但字面

量表达式在书写上更为简洁，因而推荐使用。不过，如果在执行时需要对正则表达字符串进行组合，则仍需要使用 RegExp 的构造函数调用。

通过字符串书写的正则表达式时，有一些需要注意的地方。那就通过 JavaScript 的字符串值书写反斜杠字符时，必须对其进行转义。

```
// 表示头部为空白字符的正则表达式模式
js> var reg = /^\s+/;

// 如果通过字符串来传递该模式，则需要对反斜杠字符进行转义
js> var reg = new RegExp('^\\s+');
```

表 7.15 总结了 RegExp 类的函数以及构造函数调用，表 7.16 总结了其属性。

表 7.15　RegExp 类的函数以及构造函数调用

函数或是构造函数调用	说明
RegExp(pattern, flags)	生成一个正则表达式模式为 pattern 的 RegExp 实例
new RegExp(pattern, flags)	生成一个正则表达式模式为 pattern 的 RegExp 实例

表 7.16　RegExp 类的属性

属性名	说明
prototype	用于原型链
length	值为 2

表 7.17 总结了 RegExp.prototype 对象的属性。具体的例子则将在下一节中介绍。

表 7.17　RegExp.prototype 对象的属性

属性名	说明
constructor	对 RegExp 类对象的引用
exec(string)	返回对输入字符串 string 进行正则表达式匹配的结果
test(string)	返回对输入字符串 string 进行正则表达式匹配的结果的布尔值
toSource	JavaScript 自定义的增强功能。其求值结果将返回用于生成 RegExp 实例的字符串（源代码）
toString()	将正则表达式转换为字符串形式。该字符串的格式为在 / 与 / 之间包含了相应的正则表达式

表 7.18 总结了 RegExp 类的实例属性。

表 7.18　RegExp 类的实例属性

属性名	说明
[内部值]	正则表达式模式的内部形式
ignoreCase	正则表达式的旗标之一
global	正则表达式的旗标之一
lastIndex	用于标识下一次匹配的起始位置的字符串下标
multiline	正则表达式的旗标之一
source	正则表达式模式的字符串

7.4.5　正则表达式程序设计

■ test 方法与 exec 方法

实际中会用到的正则表达式对象的方法有 exec 与 test 两种。我们首先说明 test 方法。

如果输入字符串与模式相匹配，test 方法就会返回真，如果没有得到匹配，则会返回假。具体的例子请参见前一节开头的代码示例。

exec 方法则稍有些复杂。exec 方法会在模式与输入字符串相匹配时返回一个结果对象（数组）。如果没有得到匹配则返回 null 值。因此，如果对返回值是否为 null 进行判定，则能够得到与 test 相同的结果。如果仅仅需要知道是否得到匹配，使用 test 方法效率更高。

■ exec 方法的返回值

exec 方法的返回值是一个用于表示匹配结果的数组。要理解该返回值的含义，则需要对正则表达式中的分组有一定的了解。所谓正则表达式的分组，指的是能够在正则表达式中引用匹配字符串中的子字符串（也称为前向引用）。能够被引用的子字符串称为一个分组。可以将模式内的一部分用圆括号括起来进行分组。在一个模式中，可以多次使用圆括号分组。对于前向引用来说，分组号是从 1 开始计的下标。仅通过文字说明可能有些难以理解，请看图 7.12 中的例子。

在 exec 方法的返回值中，数组的第 0 个元素是在输入字符串中第一个得到匹配的字符串，而从数组的第 1 个元素起则是分组的前向引用的子字符串。图 7.12 是没有全局旗标（请参见表 7.14）的情况。

对于正则表达式 /(\w+)\s(\w+)/ 的匹配来说，是否进行分组结果都没有区别。如果去除用于分组的圆括号，则变为了 /\w+\s\w+/。这个模式能够匹配的字符串为：一个或是更多个能够组成单词的字符，接着空白字符，再接着一个或是更多个能够组成单词的字符。简单来讲，可以理解成单词加空格加单词的形式。如果以这个模式对输入字符串 'abc def ghi jkl' 进行匹配，则开头的两个单词将得到匹配。通过正则表达式分组，就可以前向引用每一个单词。也就是说，前向引用的第一个分组是 'abc'，第二个分组则是 'def'。因此，exec 方法返回值的数组元素就是图 7.12 中的形式。

图 7.12 exec 方法的具体示例（不含全局旗标）

```
js> var text = 'abc def ghi jkl';
js> var reg = /(\w+)\s(\w+)/;  // (\w+) 用于指定分组（共两个）
js> reg.exec(text);
["abc def", "abc", "def"]
// 数组的第 0 个元素是完整的匹配字符串。从数组的第 1 个元素起是分组引用的子字符串
```

使用了全局旗标，会在多次调用 exec 方法时不断地寻找下一个匹配。从内部来看，RegExp 对象的 lastIndex 属性的值将会更新以供下一次查找时使用。

图 7.13 是一个具体的例子。如果没有找到匹配，exec 方法会返回 null，所以通常的做法是在一个循环中查找。

图 7.13 exec 方法的具体示例（有全局旗标）

```
js> var text = 'abc def ghi jkl';
js> var reg = /(\w+)\s(\w+)/g;

js> reg.exec(text);            // 第 1 次查找
["abc def", "abc", "def"]
// 数组的第 0 个元素是完整的匹配字符串。从数组的第 1 个元素起是分组引用的子字符串
js> reg.exec(text);            // 第 2 次查找
["ghi jkl", "ghi", "jkl"]
js> reg.exec(text);            // 在没有找到时会返回 null
null
```

7.4.6 字符串对象与正则表达式对象

在 String 对象中有不少以正则表达式对象作为参数的方法。表 7.19 是表 3.4 的一部分。

表 7.19 String 对象中将正则表达式对象作为参数的方法

属性名	说明
match(regexp)	返回正则表达式 regexp 的匹配结果
replace(searchValue, replaceValue)	将 searchValue（正则表达式或是字符串值）替换为 replaceValue（字符串或是函数）并返回相应的字符串
search(regexp)	返回正则表达式 regexp 的匹配位置的下标
split(separator, limit)	通过参数 separator（字符串或是正则表达式）对字符串进行分割，并返回一个字符串值数组

search 方法比较简单，它返回的是字符串中与指定的正则表达式模式相匹配的位置。如果没有匹配的话则返回 -1。

在字符串的分割（split）中，将把指定的模式作为分割字符，对字符串进行分割操作。其结果是一个字符串的数组。例如，字符串为 "abc,def,ghi" 而分割字符的模式为逗号（,）时，会将其分割成 "abc"、"def"、"ghi" 这 3 个字符串。还可以通过正则表达式模式来指定分割字符。关于 split 方法的具体示例，请参见 7.1.9 节。

在 search 与 split 方法中，正则表达式的全局旗标不起作用（会被忽略）。

而 replace 与 match 的操作则根据全局旗标的不同而有所变化。首先说明 replace。replace 的作用是将字符串中的子字符串替换为别的字符串。replace 的第 1 参数可以通过正则表达式指定。与正则表达式相匹配的部分就是将会被替换的部分。如果对正则表达式设置了全局旗标，则会替换所有与模式相匹配的子字符串；如果没有设置全局旗标，则仅会替换第一个匹配的子字符串。此外，如果在第 1 个参数的正则表达式中进行了分组，则还能在第 2 个参数的字符串中，通过符号来标识分组的前向引用。需要以 $1、$2 这样的 $ 加上数字方式来进行前向引用。关于其他一些符号，请参见表 7.20。图 7.14 是一个 replace 的具体示例。

表 7.20　replace 中支持的前向引用

前向引用	说明
$&	相匹配的字符串
$ 数字	分组的前向引用。数字从 1 开始计
$`	位于匹配之前的字符串
$'	位于匹配之后的字符串

图 7.14　replace 的具体示例

```
js> var text = 'abc def ghi jkl';

// 将空格替换为逗号字符
js> text.replace(/\s/, ',');         // 没有全局旗标
"abc,def ghi jkl"

// 对空格之前的字符进行分组，并将其移动至逗号字符之后
js> text.replace(/(.)\s/g, ",$1");
"ab,cde,fgh,ijkl"

// 如果使用第 2 参数，就能够对每一个匹配进行回调
// 回调函数的第 1 个参数是整个匹配，从第 2 个参数起是分组的前向引用
// 回调函数的返回值是替换字符串
// 在下面的例子中，替换操作的结果与前例相同
js> text.replace(/(.)\s/g, function(m0,m1) { return ',' + m1; } );
"ab,cde,fgh,ijkl"
```

match 方法将会返回元素为与模式相匹配的子字符串的数组。根据全局旗标设置与否，其返回值的结果会发生变化。在设置了全局旗标时，将会返回一个以输入字符串中所有与模式相匹配的子字符串为元素而组成的数组。如果没有设置全局旗标，则数组的第 0 个元素是第一个获得匹配的子字符串，从数组的第 1 个元素起是分组引用的子字符串。这时与 RegExp 的 exec 方法的返回值相同。

图 7.15 中是一个具体的例子。

图 7.15　match 的具体示例

```
js> var text = 'abc def ghi jkl';

// 设置了全局旗标
js> text.match(/\w/g);
["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l"]
js> text.match(/\w+/g);
["abc", "def", "ghi", "jkl"]
js> text.match(/(\w+)\s(\w+)/g);
```

```
[ "abc def", "ghi jkl" ]          // 一个元素为所有相匹配的子字符串的数组

// 没有设置全局旗标
js> text.match(/(\w+)\s(\w+)/);
["abc def", "abc", "def"]         // 数组的第 0 个元素是整个匹配字符串
                                  // 从数组的第 1 个元素起是分组引用的子字符串
```

专栏

ECMAScript 第 5 版中的严格模式

严格模式是在 ECMAScript 第 5 版中被引入的概念。其使用方式是在代码的起始行处写下以下指令。如果写在函数的起始行，则只有该函数会使用严格模式。

```
'use strict';
"use strict";
```

可以看到，这里的指令只是普通的字符串而已。所以在不支持严格模式的环境下，它被当作没有意义的语句被忽略。

在严格模式中，JavaScript 的一些语言功能将会受到限制。也就是说，在通常情况下可用的 JavaScript 代码在这时将会引起错误。通过严格模式可以避免 JavaScript 中的很多容易产生错误的地方。所以即使并不会在实际中使用，学习一下严格模式也很有意义，可以了解一些语言陷阱。

下面总结了严格模式中具有代表性的限制。

- 禁用隐式的全局变量
- 在函数内部的 this 引用不会指向全局对象
- NaN、Infinity、undefined 的全局变量是只读的
- 禁用同名的属性名
- 禁用同名的形参名
- 禁止访问 arguments.callee
- 禁止访问 Function 对象的 caller 属性
- 禁用 with 语句
- eval 不会生成新的标记



第 3 部分

客户端 JavaScript

本部分将阐述在浏览器中运行的 JavaScript，以帮助读者深入理解其特殊之处。

第 8 章 客户端 JavaScript 与 HTML

本章讲述客户端 JavaScript 的开发方法、运行方法以及调试方法。客户端 JavaScript 的开发、运行与调试并不需要专门的工具，只要有浏览器或文本编辑器就能立即开发了。

8.1 客户端 JavaScript 的重要性

8.1.1 Web 应用程序的发展

随着互联网的发展，现在的网页已经能够支持各种复杂的功能了。这里所说的网页已经不仅仅是单纯的文档，而是变为了一种应用程序，所以也称为 Web 应用程序。

■ Web 应用程序的功能

Web 应用程序会在两个地方执行操作以实现其功能，即服务器端与客户端（浏览器）。对于服务器端的处理，可以使用 Java、Perl、Python、Ruby、SQL 等多种类型的语言实现。与之相对，用于描述客户端功能的语言可以说只有 JavaScript 一种。

除了 JavaScript，能够实现客户端程序功能的技术还有 Adobe Flash 和 Silverlight，不过它们只能在特定的环境中运行。鉴于这一限制，要开发、发布能够广泛运用的 Web 应用程序，最好选择 JavaScript。

此外，JavaScript 也能够在服务器端运行，不过在本部分中不做详细介绍。在这一部分中提到的 JavaScript 指的都是客户端 JavaScript。服务器端 JavaScript 会在本书的第 6 部分中说明。

现在的 Web 应用程序已经能够提供各种各样的功能。下面列举其中一些基本功能。

- 拖曳操作（Drag and drop）
- 异步读取
- 键盘快捷键（键盘访问）
- 动画效果

这些功能基本上都可以通过 JavaScript 实现。Web 应用程序所能实现的功能正在逐渐增强，今后将提供不逊于桌面应用程序的体验。

与此同时，在以 Google Chrome 的扩展程序及 Web 应用为代表的非 Web 页面环境中，JavaScript 的应用也越来越普及。因此，通过 JavaScript 来实现的功能越来越丰富，用 JavaScript 开发的情况也越来越多。

8.1.2 JavaScript 的性能提升

虽然现在 JavaScript 在 Web 应用程序的开发中是不可或缺的一部分，但在过去它的运行速度非常缓慢，无法以令人满意的速度实现复杂功能。不过随着浏览器中所使用的 JavaScript 处理引擎的性能不断提升，现在的 JavaScript 已经能够确保以较快的速度来运行各种功能了。

而且浏览器的开发状况也变得越来越活跃。Google Chrome 或 Firefox 都是每 6 个星期升级一次，以此为发布周期进行开发。以前 Internet Explorer 6 到 Internet Explorer 7 的发布之间经过了大约 5 年的时间，对比一下，我们就能够了解现在浏览器的开发到底有多活跃了。

得益于快速的发布周期，JavaScript 处理引擎的功能得到了大幅增强，JavaScript 的性能得以提升。同时，HTML5 与 CSS3 等新技术的实现也日渐完善，已经有越来越多的操作只通过浏览器就能实现。关于 HTML5，将在之后的第 4 部分中说明。

8.1.3　JavaScript 的作用

JavaScript 的作用之一是提供良好的用户体验，使应用程序能够具有更加易于理解的界面外观以及更高的易用性。为了实现某一功能，可以有多种方法。用户所需要的是更为直观的操作方式。进一步说，他们追求的是轻松愉快的使用过程。而 JavaScript 所能实现的正是这些。

应该尽可能考虑如何利用 JavaScript 实现优秀的用户界面，但是不应该认为仅仅依靠 JavaScript 就能实现所有的功能。理由有以下两点。

- 很多浏览器都禁用 JavaScript
- 有些浏览器允许用户执行自定义的 JavaScript

也就是说，在有些情况下，并不能保证 JavaScript 能够按照 Web 应用程序开发者的预期执行。所以，应该理解 JavaScript 的使用范围与局限性，在服务器和客户端分别选择合适的实现方式。

8.2　HTML 与 JavaScript

8.2.1　网页显示过程中的处理流程

在介绍 JavaScript 之前，我们首先需要了解一下浏览器显示 Web 页面时的流程。首先来确认一下浏览器在显示以下 Web 页面时执行了哪些处理。

在下面的例子中，CSS 和 JavaScript 以单独文件的形式存在，另有一个单纯用于显示图像的 Web 页面（代码清单 8.1）。

代码清单 8.1　基本的 HTML

```
<!DOCTYPE HTML>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Sample Page</title>
    <link rel="stylesheet" type="text/css" href="/css/sample.css">
    <script src="/js/sample.js"></script>
</head>
<body>
    <img src="/image/sample.png">
</body>
</html>
```

浏览器在访问该页面时执行了以下处理。

- 分析 HTML
- 载入外部 JavaScript 文件以及 CSS 文件
- JavaScript 在分析后开始执行
- 构造 DOM 树
- 载入图像文件等外部资源
- 全部完成

这里的要点在于图像文件等内容是在构造完 DOM 树之后才下载的，因此如果在构造完 DOM 树后再执行 JavaScript，用户的等待时间就可以减少。具体内容之后会详述。

8.2.2　JavaScript 的表述方式及其执行流程

浏览器正确读取 Web 页面后就能从服务器取得 HTML 页面。之后 Web 浏览器将分析该 HTML 文件并显示画面。

为了执行 JavaScript，需要在 HTML 文件内以特定的方式书写 JavaScript 的代码。JavaScript 的书写方法有很多种，其执行的流程也各不相同。以下是记述方法一览。

- <script> 标签
- 读取外部 JavaScript 文件
- onload
- DOMContentLoaded
- 动态载入

接下来，我们开始逐一说明各种书写方法及其执行流程。

■ <script> 标签

在 <script> 标签内书写 JavaScript 是一种最为简单的方法。

这种情况下，在 <script> 标签被分析之后就会立即执行 JavaScript。需要注意的是，这样将无法操作 <script> 标签之后的 DOM 元素。

由于 JavaScript 是在分析 <script> 标签之后就立即开始执行的，而这时 <script> 标签之后的 DOM 元素还未构造，因此在 <script> 标签内就无法取得位于其后的 DOM 元素（代码清单 8.2）。

代码清单 8.2　无法操作的元素

```
<div id="a"></div>
<script>
    var a = document.getElementById('a');
    alert(a !== null); // => true
    var b = document.getElementById('b');
    alert(b !== null); // => false
</script>
<div id="b"></div>
```

为了避免这个问题，最简单的方法就是在 body 的结束标签前才书写 <script> 标签（代码清单 8.3）。这样一来，在读取 <script> 标签时其他所有的 DOM 元素都已分析，从而能够操作 HTML 文件中所有的 DOM 元素。

即使如此，还是要避免对 body 操作。这是因为这时的 <body> 标签还没有结束，如果对其操作，后果可想而知。

如果希望对 body 操作，可以通过下面的 onload 和 DOMContentLoaded 方式来执行。它们的执行流程允许对 body 操作。

代码清单 8.3　可以对所有的元素操作的方法

```
<body>
    <div id="a"></div>
    <div id="b"></div>
    <script>
        var a = document.getElementBy Id('a');
        alert(a !== null); // => true
        var b = document.getElementById('b');
        alert(b !== null); // => false
</script>
</body>
```

■ 读取外部 JavaScript 文件

虽然直接在 <script> 标签内书写 JavaScript 非常简单，但在实际的 Web 应用程序开发中，这种方式几乎不会被采用。大多数情况下，都会准备单独的 JavaScript 文件，然后从 HTML 文件中读取这些文件。这时的书写方式如代码清单 8.4 所示。

代码清单 8.4　读取外部 JavaScript 文件

```
<head>
    <script src="http://example.com/js/sample.js"></script>
</head>
```

在这种情况下将会读取文件 http://example.com/js/sampe.js 并执行。该文件将会在 <script> 标签分析之后马上读取。一旦文件读取完成，文件内的 JavaScript 就将执行。

我们还可以对 <script> 标签指定 defer 属性和 async 属性（代码清单 8.5）。通过指定 defer 属性，可以使该 <script> 标签的处理推迟至其他所有的 <script> 标签之后。而如果指定了 async 属性，则会以异步方式读取外部文件，并在读取完成后依次执行。

代码清单 8.5　defer 属性与 async 属性

```
<script src="http://example.com/js/sample1.js" defer></script>
<script src="http://example.com/js/sample2.js" async></script>
```

把 JavaScript 分离至外部文件具有很多好处。

首先，浏览器就能够缓存 JavaScript 文件了。如果 JavaScript 文件的内容变化并不频繁，只要下载一次 JavaScript 文件后将其缓存，在第二次读取时就能够避免不必要的下载，直接提高运行速度。

其次，HTML 与 CSS 和 JavaScript 文件分离之后，团队分工将变得更加容易。例如，HTML 和 CSS 主要由负责界面设计的人员来书写，而 JavaScript 则由负责实现功能的人来书写。

进一步来说，一般的编辑器对于某一特定文件只能采用一种文法高亮显示模式。也就是说，如果 HTML 和 CSS 与 JavaScript 都书写在同一个文件中，就可能会因为没有通用的文法高亮模式而导致无法实现文字色彩的区分，从而导致可读性的下降。

为了避免产生这种情况，最好将代码分别写在不同的文件里。

■ onload

如果在 onload 事件处理程序（event handler）中书写 JavaScript 代码，则能够在页面读取完成后再对其执行。因为在执行时已经完成了整个页面的读取，所以可以对所有的 DOM 元素操作。关于事件处理的具体内容将在之后说明。

如果要直接写在 HTML 内，可以像代码清单 8.6 这样将其写在 <body> 标签里。如果要写在外部的 JavaScript 文件内，则可以像代码清单 8.7 这样书写。

代码清单 8.6　onload 事件处理程序

```
<body onload="alert('hello')">
```

代码清单 8.7　在外部 JavaScript 文件中使用 onload 事件处理程序

```
window.onload = function () { alert('hello'); };
```

必须注意的是，如果使用了 onload 事件处理程序，则会在读取了所有写在 HTML 文件中的图像文件之后才对其执行。因此，如果在页面内存在大型的图像文件，就可能需要花费很多不必要的时间等待图像的读取后才开始执行 JavaScript。

如果无需通过 JavaScript 对图像处理，或图像的处理内容与其体积无关，则没有必要等整个图像读取之后再开始处理。在载入图像的同时执行 JavaScript 处理的话就能够缩短用户的等待时间。

■ DOMContentLoaded

对于使用上述的 onload 方法，执行 JavaScript 时可能需要一定的等待时间，这可以使用 DOMContentLoaded 来解决。DOMContentLoaded 是在完成 HTML 解析后发生的事件。将事件侦听器设置为对该事件侦听，就能够减少执行 JavaScript 之前的不必要的等待时间（代码清单 8.8）。

代码清单 8.8　对 DOMContentLoaded 事件侦听

```
document.addEventListener('DOMContentLoaded', function () {
    alert('hello');
), false);
```

不过 DOMContentLoaded 也存在一些问题，就是它在 Internet Explorer 8 之前的浏览器中是不受支持的。不过，也有方法能够在早期的 Internet Explorer 中实现相同的功能。

具体来说，就是等到 doScroll() 方法不再抛出异常之后才开始执行 JavaScript 部分（代码清单 8.9）。这里利用的原理是，在 DOM 树的构造过程中执行 doScroll() 方法就将会引发错误。

代码清单 8.9　在 IE 中模拟 DOMContentLoaded 事件

```
function IEContentLoaded(callback) {
    (function () {
        try {
            document.documentElement.doScroll('left');
        } catch(error) {
            setTimeout(argument.callee, 0);
            return;
        }
        callback();
    }());
}
IEContentLoaded(function () {
    alert('hello');
});
```

■ **动态载入**

在 JavaScript 中，我们可以在生成 script 元素过程中动态地载入 JavaScript 文件（代码清单 8.10）。

代码清单 8.10　JavaScript 的动态载入

```
var script = document.createElement('script');
script.src = 'other-javascript.js';
document.getElementsByTagName('head')[0].appendChild(script);
```

在使用这种方法执行 JavaScript 时，JavaScript 文件在下载过程中并不会阻断其他的操作。这是一个较大的优点。如果直接在页面内书写 script 元素，则在下载该 JavaScript 文件的过程中，其他图像文件或 CSS 文件的下载将被阻断。不过，只要使用这种动态的载入方式，就能够避免下载被阻断而继续处理。

8.2.3　执行流程的小结

在考虑选用何种 JavaScript 执行流程时，DOMContentLoaded 是最为恰当的选择。尽管在 Internet Explorer 中实现同样的功能需要花费一些功夫，不过如果不考虑这个问题，可以说 DOMContentLoaded 是最佳的实现方式。

8.3　运行环境与开发环境

8.3.1　运行环境

要说起人们最熟悉的 JavaScript 运行环境，答案自然是浏览器。有以下这些著名的浏览器：Internet Explorer、Mozilla Firefox、Google Chrome、Safari 以及 Opera。

8.3.2　开发环境

编写 JavaScript 代码时，必不可少的书写工具只有以 Emacs 或 Vim 等为代表的文本编辑器。Eclipse 或 NetBeans 等多用途 IDE 也具有 JavaScript 开发所需的相关功能，使用这些进行开发也是不错的选择。此外还有 WebStorm 这样的 JavaScript+HTML 开发专用的 IDE，不过它是一款收费软件。

8.4 调试

在编写程序的过程中，调试是不可避免的一个环节。特别是最近大型 JavaScript 程序的开发越来越多，在编写程序时考虑调试难易度的问题也变得越来越重要。在此我们将说明一下 JavaScript 的调试方法。

8.4.1 alert

在 JavaScript 代码中加上 alert 语句是一种简单的调试方法。在打开浏览器时将会显示 alert 对话框。可以说，这是所谓的 printf 调试方式的 JavaScript 版本。

这种调试方法在任何浏览器中都可以实现。在显示 alert 对话框时，所有的 JavaScript 处理都会中止。因此，如果添加了大量的 alert 语句，就能够实现单步执行。不过这时需要一次次去关闭 alert 对话框，并不方便。而且，如果不小心无限循环执行了 alert 语句，就不得不强制结束浏览器的进程以关闭窗口。必须多加注意。

此外，还可以像代码清单 8.11 这样，通过覆盖对象的 toString() 方法来修改 alert 对话中显示的字符串。

代码清单 8.11　修改 alert 所显示的字符串

```
var Foo = function (text) {
    this.text = text;
};

var Foo = new Foo('hello, alert.');
alert(foo);       // => 将显示 [object Object]
foo.toString = function () {
    return this.text;
};
alert(foo);       // => 将显示 Hello, alert.
```

8.4.2 console

最近的一些浏览器内置了用于运行 JavaScript 的控制台功能。早先只有 Firefox 的 Firebug 插件才提供了这样的功能。由于使用 Firebug 进行开发的开发者数量众多，因此 Safari 以及 Google Chrome 等浏览器中也开始内置这一功能。

在 JavaScript 代码内部写上 console.log('foo bar') 语句之后，就能够在控制台中显示 foo bar 了。这在本质上与使用 alert 是一样的，不过因为在使用 console 时不需要一次次去关闭对话框，所以会比使用 alert 更为方便。同时，console 中也能够显示比 alert 方法更为详细的信息。

但是在 Internet Explorer 等一些没有内置 console 对象的浏览器中，这种方法自然就会引起错误。

■ 虚拟 console 对象

按理说，在实际运行应用程序时应该删去所有 console 相关的代码。不过在开发过程中常常会需要对程序进行一些测试，如果为此每次都删去 console 部分的代码，未免有些麻烦。于是，为了在 Internet Explorer 这类不支持 console 对象的浏览器中也能够正常运行程序，有时会采用嵌入虚拟 console 对象的做法。

可以像代码清单 8.12 这样，生成一个含有 Firebug 1.7.2 中的 console 对象所支持的所有方法的虚拟 console 对象。只要在首先载入的 JavaScript 起始处添加该对象，就能够避免在调用 console 时发生错误。在实际发布程序时，这部分的代码应该和 console.log 等内容一起删去。当然，即使留在代码中也没什么。不过考虑到应该尽可能减少代码量以提高性能，还是应该将它们删去。

代码清单 8.12　虚拟 console 对象

```
if (!window.console) {
```

```
    (function (win) {
        var names = [
            'assert', 'clear', 'count', 'debug', 'dir', 'dirxml',
            'error', 'exception', 'group', 'groupCollapsed', 'groupEnd',
            'info', 'log', 'notifyFirebug', 'profile', 'profileEnd',
            'table', 'time', 'timeEnd', 'trace', 'warn'];
        var consoleMock = {};
        for (var i = 0, len = names.length; i < len; i++) {
            consoleMock[names[i]] = function () {};
        }
        win.console = consoleMock;
    }(window));
}
```

■ 显示消息及对象

下面的方法具有的功能基本相同，它们都会从参数中取得消息或对象，然后将取得的内容输出至控制台。这些方法可以指定多个参数，且每一个指定的参数都将输出。

- console.log()
- console.warn()
- console.debug()
- console.info()
- console.error()

一般来说，仅使用 log() 方法就能够满足需要。而如果使用 debug() 方法，则可以知道输出的内容是在 JavaScript 文件中的哪一行。当需要使用大量 console.log() 类型方法时，选用 debug() 方法会很方便。

error()、warn()、info() 也会显示行数。它们与 debug() 的区别仅仅在于所显示的图标与文字的色彩会有所差异，因此在平时的使用中不必太在意这些。

此外，如果对第 1 个参数指定格式，则会对从第 2 个参数起的对象应用该格式（代码清单 8.13）。如果想要提高日志消息的易读性，就可以利用这一方法。

代码清单 8.13 格式

```
console.log('%s is %d.', 'The Answer to (the Ultimate Question of) life, the
universe, and everything', 42);
// => The Answer to (the Ultimate Question of) life, the universe, and everything is 42.
```

■ 分析对象并显示

console.dir() 方法可以完整输出作为参数接收到的对象，并将其一目了然地显示出来。console.dirxml() 方法可以将 DOM 元素以 HTML 的形式显示。这两种方法的输出内容都比 console.log() 等方法更为清楚，但也会让日志的篇幅变得很长，在使用时应加以注意，不要因此而忽略了其他的日志内容。

■ 显示栈追踪

使用 console.trace() 方法就可以显示该函数的调用者，并可以据此了解具体是哪一个对象的哪一个事件触发了该函数。

在事件驱动的程序设计过程中，要清楚地知道某个函数在什么时候什么位置调用并不是一件容易的事。这时就可以使用 console.trace() 方法来理清各种调用关系，非常方便。

■ 测量时间、次数与性能

我们可以通过 console.time() 方法与 console.timeEnd() 方法测量这两个方法之间经过的时间（代码清单 8.14）。其参数分别为每次计时的名称。具有相同名称的方法将会配对，其间经过的时间则会输出。时间的显示单位为毫秒。

代码清单 8.14 console.time()

```
console.time('foo');
alert('foo 计时开始 ');
```

```
console.time('baz');
alert('baz 计时开始 ');
console.timeEnd('baz');
alert('baz 计时结束 ');
console.timeEnd('foo');
alert('foo 计时结束 ');
```

使用 console.count() 就可以知道该行具体执行了多少次（代码清单 8.15）。

代码清单 8.15　console.count()

```
for (var i = 0; i < 100; i++) {
    console.count('foo');
    if (i % 10 === 0) {
        console.count('bar')
    }
}
```

```
foo: 100
bar: 10
```

此外，如果使用 console.profile() 或 console.profileEnd()，则可以获得更为详细的测量结果。它们可以用于获取各个函数分别执行了多少次，或总的执行时间等信息。因为在性能分析的过程中，这一分析也会使用 CPU 资源，所以最终得到的结果将会比正常情况更差。不过另一方面，在性能优化的过程中，哪些函数被多次调用并花费了大量的执行时间是很重要的信息，所以这一功能仍然是不可获缺的。

■ 使用断言

console.assert() 的功能是仅在指定条件为 false 时输出日志。例如，为了确认一个不能接收 null 的参数没有收到 null 值，就可以像下面这样使用该方法（代码清单 8.16）。

代码清单 8.16　console.assert()

```
function foo(notNullObj) {
    console.assert(notNullObj != null, 'notNullObj is null or undefined');
    // 其他代码
}

foo(1);        // => 没有显示断言。调用正确
foo(null);     // => 显示了断言。调用错误
foo();         // => 显示了断言。调用错误
```

如果某一函数在参数为 null 时会发生错误，则可以通过断言来检查。

8.4.3　onerror

在 JavaScript 中，如果发生了错误，则会执行 Window 对象的 onerror 属性所指向的函数。这个函数接受 3 个参数。第 1 个参数是错误消息，第 2 个参数是包含了发生错误的 JavaScript 文档的 URL，第 3 个参数是发生错误位置的行数。此外，如果该函数的返回值为 true，则浏览器将不会执行输出错误日志这一默认行为。

在开发过程中进行测试时，可以通过 onerror 事件句柄输出专门的日志并将其发送给服务器。这样就能获得一些额外的调试信息。

8.4.4　Firebug, Web Inspector (Developer Tools), Opera Dragonfly

最近的浏览器不少都内置了能够图形化显示 DOM 内容或与服务器的通信内容等信息的开发工具。Firefox 的插件 Firebug 是这一类型的开发工具的开创者。之后出现的开发工具大都实现了和 Firebug 等同的功能。下面是一些经常使用的功能。

- HTML/CSS 的内容确认与编辑
- JavaScript 控制台
- JavaScript 调试工具
- JavaScript 性能分析工具
- 网络监控

在最近的 Web 开发中，客户端的处理工作增加了很多。上面这些功能也变得越来越不可或缺。应该积极地利用这些功能，尽可能地发挥它们的作用。

■ JavaScript 调试工具

通过 JavaScript 调试工具，可以轻松地在代码中设置断点以从断点处开始单步执行，或监视某一变量的状态。

如果在源代码中写有 debugger，则可以在运行至此处时启动调试工具（代码清单 8.17）。这就如同预先在源代码中设置了断点一样。

代码清单 8.17　从源代码中启动 JavaScript 的调试工具

```
function foo() {
    // ……任意的处理
    debugger;     // 运行至此时调试工具将会启动
    // ……任意的处理
}
```

■ 使用 JavaScript 性能分析工具时的一些注意点

性能分析工具的功能是测量并显示某一函数在运行时需要花费的时间。这对于优化 JavaScript 代码并提高运行性能来说是不可缺少的。

在使用性能分析工具时，必须要注意匿名函数（anonymous function）的情况。在 JavaScript 中，函数作为第一类对象使用，所以不必对函数命名也能执行该函数。像代码清单 8.18 这样注册事件侦听器的方式是很常见的。

代码清单 8.18　用于匿名函数的事件侦听器

```
foo.addEventListener('click' function (event) {
    // foo 在发生点击事件时将会进行的处理
}, false);
```

这里所写的

```
function (event) {}
```

就是一个匿名函数。

在性能分析工具中，匿名函数将显示为 anonymous function。如果只用了一个匿名函数，并不会与其他函数产生混淆，自然也就没有问题。但是在使用了多个事件侦听器时，就无法判断具体是哪一个函数了。

为了避免这样的问题，可以改用代码清单 8.19 或代码清单 8.20 这样的书写方式。这样一来，匿名函数就不再是 anonymous function，而成为了一个具有名称的函数，而性能分析工具也就能够正确地识别并表示该函数。

代码清单 8.19　用于具有名称的函数的事件侦听器

```
// 将其命名为 bar
foo.addEventListener('click', function bar(event) {
    // foo 在发生点击事件时将会进行的处理
}, false);
```

代码清单 8.20　用于具有名称的函数的事件侦听器

```
// 另外定义一个名为 bar 的函数
```

```
function bar(event) {
    // foo 在发生点击事件时将会进行的处理
}
foo.addEventListener('click', bar, false);
```

对于上面两种书写方式，所有的开发工具都会将其作为具有名称的函数来显示。

不过，如果像代码清单 8.21 这样书写，Internet Explorer 8 的开发者工具无法将其识别为具有名称的函数。该函数依然会显示为 anonymous function。

代码清单 8.21　用于 Function 对象的事件侦听器

```
// 准备一个名为 bar 的 Function 对象
var bar = function (event) {
    // foo 在发生点击事件时将会进行的处理
}
foo.addEventListener('click', bar, false);
```

■ 网络监控

在开发大量使用 AJAX 的网页时，通过使用网络监控功能，可以确认各个请求是否已正确发送。各个请求的请求头部、请求参数、报文数据、响应头部、响应体等大部分与该请求相关的数据都能够通过这一功能获取。

另外，该工具也能够对用于测量组成网页的 HTML 文件、图像文件、JavaScript 文件等资源在载入过程中需要花费多少时间。

8.5 跨浏览器支持

如果要通过多种不同的浏览器访问页面，则书写 JavaScript 以及 CSS 时必须多加注意。

HTML 渲染引擎与 JavaScript 引擎等共同组成了一个浏览器。HTML 渲染引擎将会对 HTML 与 CSS 进行解析，并以恰当的形式显示解析的结果。而 JavaScript 引擎的功能则是解析 JavaScript 并执行相应操作。

这里存在的问题是，主流的浏览器各自采用了不同的 HTML 渲染引擎与 JavaScript 引擎。因此，即使访问的是同一个网页，根据使用的浏览器的不同，页面的显示与逻辑操作上也会有所不同。表 8.1 总结了各种浏览器所使用的引擎。

表 8.1　各种浏览器所使用的引擎

浏览器	渲染引擎	JavaScript 引擎
Internet Explorer	Trident	Jscript
Mozilla Firefox	Gecko	SpiderMonkey
Google Chrome	WebKit[①]	V8
Safari	WebKit	JavaScriptCore
Opera	Presto[②]	Charakan

所谓跨浏览器支持，指的就是如何在使用这些不同的浏览器访问页面时能尽可能地保持一致的页面显示与行为的方法。

8.5.1　应当提供支持的浏览器

如今，世界上已经有各种各样的浏览器。从性价比的角度来看，要支持所有的浏览器是没有实际意义的。要支持哪些浏览器，取决于用户正在使用的浏览器都有哪些。单纯考虑浏览器的占有率的话，只要

① Google 于 2013 年 4 月宣布 Chrome 将会在今后使用 xx 渲染引擎。——译者注

② Opera 于 2013 年 3/4 月宣布 Opera 将会在今后使用 WebKit 渲染引擎。——译者注

对以下浏览器提供支持即可。

- Internet Explorer 6 及其后继版本
- Google Chrome（最新版）
- Firefox（最新版及其前一版本）
- Safari（最新版）
- Opera（最新版）

然而，浏览器的占有率是不断变化的，每次都要支持多种浏览器也需要花费一定的开发成本。例如，如果仅在公司或学校的系统内使用的页面，在某种程度上来讲，所使用的浏览器往往是能够确定的。这时，不考虑对设想之外的浏览器提供跨浏览器支持也没有什么问题。

在应当提供支持的浏览器中，Internet Explorer 是比较麻烦的一种。在 Internet Explorer 中提供了很多 ECMA-262 不支持的独有的功能。对于 Internet Explorer 的跨浏览器支持问题，已经有很多开发者论述过，能够找到很多的相关信息。因此，如果仅仅想要实现同样的功能，并不会有太大问题。不过，有时 JavaScript 可以具有很复杂的功能，这时要实现其功能则可能会是一个问题。

其中 Internet Explorer 6 是一个特别麻烦的问题。与现在最新的浏览器相比，它处理 JavaScript 的速度非常慢。如果在制作页面时发现处理速度过慢将会引起问题的话，可以干脆明确表示该页面不支持 Internet Explorer 6。事实上，Internet Explorer 9 已经开始逐步遵循 Web 标准，所以如果只考虑支持最新版的浏览器的话，可以说跨浏览器支持的问题就几乎不存在了。

8.5.2 实现方法

接下来，我们来说明支持多种浏览器的代码的书写方法。虽然只需要根据浏览器的种类与版本进行条件分支处理即可，不过具体来说应该以什么作为条件才合适还得再考虑才行。这里大致有两种分类方法。

即根据用户代理来进行区分，以及根据功能是否被支持来进行区分。这两者看似差不多，但实质并不相同。

■ 根据用户代理来判断

所谓用户代理，指的是用于识别客户端应用程序的字符串。这并不是一个仅在浏览器领域使用的术语，在搜索引擎的网络蜘蛛以及其他一些系统中也会使用。不过在这里所使用的用户代理特指浏览器的用户代理。

可以通过 Navigator 对象的 userAgent 属性来取得用户代理。该值通常是由浏览器的种类、版本、操作系统的种类及其类型决定的。在取得该值之后，就可以由该值对操作进行区分以实现跨浏览器支持。

虽然在 Navigator 对象中也有 appName、appCodeName、appVersion 这样的属性，不过使用这些值是无法正确判断浏览器的。例如对于 Firefox 4 来说，上述的属性值如表 8.2 所示。

表 8.2 Navigator 对象的属性的值

属性	值
appName	Netscape
appCodeName	Mozilla
appVersion	5.0 (Windows)

出现这种情况的原因，是为了让一些在过去使用这些值进行浏览器判断的页面在现在的浏览器上也能够正常地被访问。虽然确保兼容性也很重要，但这样的做法也造成了如今无法再通过这些属性值对浏览器进行正确判断的局面。现在，如果要正确地识别浏览器，就只能依靠 userAgent 的值了。

Firefox4 的 userAgent 的值如下所示。通过该值就能够正确地对浏览器的种类与版本等信息进行判别。

```
Mozilla/5.0 (Windows NT 6.1; WOW64; rv:2.0.1) Gecko/20100101 Firefox/4.0.1
```

表 8.3 列举了一些具有代表性的浏览器的用户代理。这里使用的操作系统是 64 位的 Windows7[①]。

① 由于版本更新，这里的浏览器可能并不是其最新版。读者可以自己确认最新版的用户代理。——译者注

表 8.3　用户代理的值

浏览器	用户代理
Internet Explorer 9	Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; WOW64; Trident/5.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0; .NET 4.0C; .NET4.0E)
Firefox 4	Mozilla/5.0 (Windows NT 6.1; WOW64; rv:2.0.1) Gecko/20100101 Firefox/4.0.1
Safari 5	Mozilla/5.0 (Windows; U; Windows NT 6.1; ja-JP) AppleWebKit/533.21.1 (KHTML, like Gecko) Version/5.0.5 Safari/533.21.1
Google Chrome 12	Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/534.30 (KHTML, like Gecko) Chrome/12.0.742.92 Safari/534.30
Opera 11.11	Opera/9.80 (Windows NT 6.1; U; js) Presto/2.8.131 Version/11.11

一般来说，用户代理会根据操作系统的不同而不同。跨浏览器支持中通常只需考虑浏览器的类型以区分。对于同一种浏览器来说，并不会因为操作系统的不同而具有不同渲染引擎及 JavaScript 引擎。不过偶尔也会遇到因操作系统不同而导致渲染出现差异，对此也请加以注意。

可以像代码清单 8.22 这样，利用所取得的用户代理值对处理分类。

代码清单 8.22　基于用户代理的跨浏览器支持策略

```
// 跨浏览器支持的事件侦听注册方法
var addEvent(target, name, fn) = function() {
    // 判断正在访问该页面的浏览器是否是 Internet Explorer
    var isIE = navigator.userAgent.indexOf('MSIE') > 0;
    if (isIE) {
        // 在 IE 中不存在 addEventListener() 方法，因此要换用 attachEvent() 方法
        addEvent = function(target, name, fn) {
            target.attachEvent('on'+name, fn);
        };
    } else {
        // 如果不是 IE，则使用 addEventListener() 方法
        addEvent = function(target, name, fn) {
            target.addEventListener(name, fn, false);
        }
    }
    addEvent(target, name, fn);
}
```

■ 根据功能是否被支持来判断

在通过用户代理来进行分支处理时，必须事先知道该用户代理是否支持某一功能（能否使用某一函数）。此外，虽然对于过去的浏览器来说，不必担心今后会再增加新功能，所以这也就没有什么问题，但现在还在继续开发中的浏览器则必定会不断地增加新的功能。

也就是说，随着浏览器的版本升级，会增加新的功能，这可能需要修改源代码。另外，根据浏览器的不同，有时用户代理可以自由修改（有时即使浏览器不支持这一功能，也可以通过扩展程序来实现用户代理的修改）。虽然一般来说不必考虑支持用户代理被改写的情况，但如果希望在这种情况下也能够正常执行操作的话，显然仅凭用户代理是不足以作为分支处理的条件的。对于通过用户代理进行条件分支判断的情况，确实可能会有类似问题产生。

只要用户代理不能够确保在将来也能够使用某些功能，这种分支处理方式就不如直接通过判断功能是否被支持来得可靠。例如，对于前一节中所提到的跨浏览器事件侦听注册方法，在支持 addEventListener() 方法的情况下使用 addEventListener() 方法，在支持 attachEvent() 方法的时候则使用 attachEvent() 方法，会是一种更为直观的判断方式。这时可以像代码清单 8.23 这样书写。

代码清单 8.23　基于功能测试的跨浏览器支持策略

```
// 跨浏览器的事件侦听注册方法
var addEvent(target, name, fn) = function () {
    if (window.addEventListener) {
        // 如果支持 addEventListener() 方法则使用该方法
```

```
        addEvent = function (target, name, fn) {
            target.addEventListener(name, fn, false);
        };
    } else if (window.attachEvent) {
        // 如果支持 attachEvent() 方法则使用该方法
        addEvent = function(target, name, fn) {
            target.attachEvent('on'+name, fn);
        };
    }
    addEvent(target, name, fn);
}
```

■ 通过用户代理判断与通过功能测试判断

通常无论使用哪一种方式，都能够正确无误地判断分支处理。不过正如之前所讲，与通过用户代理进行分支判断相比，对功能是否被支持进行判断是一种更可靠的做法，所以一般来说，应该选择根据功能是否被支持来进行条件分支处理。

不过也不是说不再使用用户代理信息进行判断了，而是说将使用用户代理的情况限定于希望在特定浏览器的特定版本中进行特别的处理时。例如，有时问题的原因并不是函数的支持情况不同，而是 CSS 解析等渲染引擎的行为差异。具体来说，在 IE6 和 IE7 中常常需要进行分支处理以执行不同的操作。这时就可以通过用户代理的值来实现分支处理。

基于用户代理的条件判断还有一个优点，那就是只需要对情况判断一次即可。在之前的例子中仅仅设定了事件侦听的注册方法，在其他的方法中也需要定义同样的分支判断。如果能够通过用户代理明确知道哪些方法是被支持，就能够只通过用户代理判断情况，而不需要一次次地判断功能是否被支持了。不过话虽如此，每次都判断功能也不会浪费多少时间，所以通常来说只需要通过功能是否被支持来判断即可。

8.6 Window 对象

在客户端 JavaScript 中，Window 对象是一个全局对象。Window 对象即对应于正被浏览器显示的窗口的一种对象。

Window 对象是 JavaScript 所能操作的最高层的对象。

以下这些是 Window 对象所具有的一些属性。

- navigator
- location
- history
- screen
- frames
- document
- parent, top, self

8.6.1 Navigator 对象

Window 对象的 navigator 属性是一个 Navigator 对象。Navigator 对象包含了浏览器的版本、浏览器所支持的插件、浏览器所使用的语言等各种与浏览器相关的信息。

在 Navigator 对象所包含的信息中，最常用的是 userAgent 属性。可以通过 userAgent 属性来实现对浏览器的识别，这在跨浏览器支持中是必需步骤。关于 userAgent 属性的使用，请参见 8.5 节。

8.6.2 Location 对象

Window 对象的 location 属性是一个 Location 对象。Location 对象包含了与当前显示的 URL 相关的一些信息。

■ href 属性

通过 href 属性，可以引用现在正在显示的页面的完整 URL。该 URL 与地址栏中所显示的字符串相同（不过在最近的一些浏览器中，协议以及查询参数等信息会隐藏，在这种情况下两者之间可能会有所不同）。

如果将 hred 属性设置为新的值，则会跳转至其他页面。

```
location.href = 'http://foobar.example.com';
```

尽管 href 属性已经包含了完整的 URL，在 Location 对象中仍有其他一些属性分别储存了协议、主机名等 URL 的各个元素。如果将这些属性的值相结合，则可以得到 href 属性的值。表 8.4 总结了 Location 对象的属性 ①。

表 8.4　Location 对象的属性

属性名	说明	示例
protocol	协议	http:
host, hostname	主机名	example.com
port	端口	8080
pathname	路径	/foo
search	查询参数	?q=bar
hash	哈希令牌	#baz

■ assign() 方法

可以使用 assign() 方法从当前页面跳转至另一页面。这与设置 href 属性的值的效果是一样的。

```
location.assign('http://foobar.example.com');
```

■ replace() 方法

通过 replace() 方法也能从当前页面跳转至另一页面。

```
location.replace('http://foobar.example.com');
```

在执行 replace() 方法后，与之前设定 href 属性的值一样，都会发生页面跳转。不同的是，两者对浏览器历史记录的处理有所差别。对 href 属性进行改写后浏览器的历史记录中会有所反映，可以通过后退键返回之前的页面。

与之相对，replace() 方法不会在历史记录中留下信息，因此无法通过后退键返回之前的页面。在理解了两者对历史记录的处理方式之后，就可以根据情况选择对 href 属性改写还是使用 replace() 方法了。

■ reload() 方法

可以通过 reload() 方法刷新当前正在显示的页面。而根据传给参数的 boolean 值的不同，刷新的方式也会有所不同。当传递 true 时将会忽略浏览器的缓存，强制重新载入数据。如果传递的是 false，则会利用浏览器的缓存来刷新页面。

```
location.reload(true);          // => 忽略浏览器缓存刷新
location.reload(false);         // => 利用浏览器缓存刷新
location.reload();              // => 与 location.reload(false); 相同
```

8.6.3　History 对象

Window 对象的 history 属性是一个 History 对象。

可以通过 History 对象的 back() 方法与 forward() 方法实现浏览器历史记录中的后退与前进。这与按下浏览器的后退键和前进键时的操作相同。

此外，go() 方法也能够实现同样的功能。go() 方法可以以参数所接收到的整数值为单位进行前进或后退操作。如果传递的是正值，则前进，如果传递的是负值，则后退。

```
history.back();                 // 在历史记录中后退
history.forward();              // 在历史记录中前进
```

① 例子中的 URL 是 http://example.com:8080/foo?q=bar#baz

```
history.go(1);              // 在历史记录中前进一次。与history.forward();相同
history.go(-2);             // 在历史记录中后退两次。
```

8.6.4 Screen 对象

Window 对象的 screen 属性是一个 Screen 对象。

Screen 对象包含了画面的大小与发色数等信息。通过这些值，能够针对大尺寸的显示器和小尺寸的屏幕分别显示不同的画面。

虽然还能够实现将窗口移动至画面中央的功能，不过随意移动窗口通常会招致用户的反感，因此必须加以注意。在最近的客户端 JavaScript 开发中，几乎没有什么能够有效利用 Screen 对象的情况。

8.6.5 对 Window 对象的引用

■ window 属性

可以通过 window 属性获取对 Window 对象的引用。window 属性所指向的 Window 对象在 JavaScript 中是一个全局对象。

也就是说，在客户端 JavaScript 中所操作的所有函数与对象都是 window 属性所引用的这个对象的属性。

■ frames 属性

当窗口中含有多个框架时，frames 属性中将会含有这些框架的引用。如果没有框架，frames 属性中则是一个空的数组。可以通过 <frameset> 标签、<frame> 标签，或 <iframe> 标签来生成一个框架。无论通过哪种方式生成，JavaScript 都会以同样的方式对其引用。

而框架本身也是一种 Window 对象，因此可以以

```
window.frames[1].frame[2]
```

这样的方式取得某个框架中的另一框架（以下称为子框架）的引用。

■ self 属性

可以通过 self 属性对 Window 对象自身引用。self 属性与 window 属性一样，引用了一个 Window 对象。

■ parent 属性

可以通过 parent 属性从子框架处取得其父框架的引用。如果不存在父框架，parent 属性则是对当前 Window 对象自身的一个引用。

■ top 属性

如果要在框架层层嵌套的情况下获取最上层的框架的引用，可以使用 top 属性。如果自身就是最上层的框架，top 属性则是对当前 Window 对象自身的一个引用。

总之，如果自己就是最上层的 Window 对象的话，window、self、parent、top 都将是对同一个 Window 对象的引用。

8.6.6 Document 对象

Window 对象的 document 属性是一个 Document 对象。关于 Document 对象的详细信息，将在第 9 章中说明。

对 Cookie 的操作是 Document 对象中不属于 DOM 范畴的一种功能。尽管通过 HTML5 的 Web Storage 或 Indexed Database，浏览器自身也能够保存一定的数据，但在没有实现这些功能之前 Cookie 是唯一一种可以保存数据的方式。可以通过 Document 对象的 cookie 属性对 Cookie 进行读写操作。

第 9 章 DOM

JavaScript 借助 DOM（Document Object Model，文档对象模型）对 HTML 进行操作。利用 DOM 这种标准方式，无论哪种浏览器都可以以同样的方式来操作 HTML 文档。只要理解 DOM，就能够自由地操作 Web 页面。

9.1 DOM 的定义

在希望通过 JavaScript 来对页面的内容进行操作时，如果文档的内容与结构能够以一种便于程序处理的形式表现，无疑会给处理带来方便。比如说，虽然可以直接修改以字符串形式表示的 HTML 资源内容，但如果可以对“id 是 foo 的 <div> 标签”，或“所有的 <a> 标签”这样的集合进行操作，则程序的可读性会更高，书写也较为简单。为此，JavaScript 中引入了 DOM 的概念。

DOM 是一种 API，其作用为在程序中使用 HTML 文档以及 XML 文档。在 DOM 中，HTML 文档与 XML 文档会以树形对象集合的形式被使用。这一树形结构称为 DOM 树。

DOM 树中的一个个对象被称为节点。节点之间形成了一个树形结构，树中的某个节点可能会引用另外一个节点。根据引用关系，分别有父节点、子节点、兄弟节点、祖先节点、子孙节点等类型。

根据 W3C 的定义，DOM 可以分为 Level1 ～ 3 这几层。

9.1.1　DOM Level 1

DOM Level 1 是由 Core 与 HTML 这两个模块组成的（表 9.1）。在 DOM Level 1 Core 中包含很多操作 DOM 树的方法。表 9.2 介绍了 DOM Level 1 Core 中定义的一些基本方法，更为详细的内容会在之后进一步说明。

表 9.1　DOM Level 1 的模块一览

模块	说明
Core	对包括 HTML 在内的基本 DOM 操作提供支持
HTML	对一些专用于 HTML 文档的方法提供支持

表 9.2　DOM Level 1 Core

方法名	说明
getElementsByTagName	根据指定的标签名来获取元素
createElement	创建新元素
appendChild	插入元素

9.1.2　DOM Level 2

与 DOM Level 1 相比，DOM Level 2 中包含了很多模块。其中包括 Events 这样与 addEventListener() 等事件处理方法相关的模块，或者 DOM Level 1 中 Core 模块以及 HTML 模块的扩展模块。

CSS 也是在 DOM Level 2 中定义的模块。表 9.3 列举了 DOM Level 2 包含的模块。可惜的是，

Internet Explorer 8 以及更早的版本并没有遵循 DOM Level 2 标准。而 Firefox 或 Google Chrome 等现代浏览器则几乎完全支持 DOM Level 2。

表 9.3　DOM Level 2 所包含的模块一览

模块	说明
Core	Level 1 Core 的扩展
HTML	Level 1 HTML 的扩展
Views	对与文档显示状态相关的功能提供支持
Events	对捕获、冒泡、取消等事件系统提供支持
Styles	对与样式表相关的功能提供支持
Traversal and Range	对 DOM 树的遍历以及范围的指定提供支持

9.1.3　DOM Level 3

DOM Level 3 是由表 9.4 中所示的模块所组成的，其中 Events 模块还没有达到推荐使用的程度。不过在现代浏览器中，Event 模块中所定义的一些内容已经提前实现。

表 9.4　DOM Level 3 所包含的模块一览

模块	说明
Core	Level 2 Core 的扩展
Load and Save	对文档内容的读取与写入提供支持
Validation	对文档内容合法性的验证提供支持
XPath	对 XPath 相关的功能提供支持
Events	Level 2 Events 的扩展。对键盘事件提供了支持

专栏

DOM Level 0

在 DOM 标准制定之前，各种浏览器所采用的对象模型被称为 DOM Level 0。DOM Level 0 也称为传统 DOM。虽然 DOM Level 0 并不能算是一种被正确定义的标准，但为了与过去的浏览器相兼容，现在的浏览器中仍然支持 DOM Level 0 中的功能。DOM Level 0 中含有 Window、Document、Navigator、Location、History 等对象。不过，在 Document 对象中也有一些在 DOM Level 1 中被定义的 API，所以不能说以上这些对象都是属于 DOM Level 0 的内容。

此外，虽然 DOM Level 0 并不能算是一种标准，但是它所包含的一些对象却是符合 HTML5 标准的。很多过去由浏览器开发商自主实现的功能现在也都成为了标准。还有一些功能虽然还没有成为标准，但也已经处于标准起草阶段了。这些由浏览器开发商自主实现的功能为了不与 DOM 标准中所定义的属性名或名称冲突，常常会根据开发商的不同分别在名称前加上前缀。其中的很多被用于 CSS 的属性或 JavaScript 的函数名中。例如，在 Firefox 中使用了 moz 这一前缀，而在 Google Chrome 与 Safari 所使用的 WebKit 中则使用了 webkit 这一前缀。

9.1.4　DOM 的表述方式

可以通过下面的方式书写 DOM。

```
接口名 . 方法名 ()
```

```
接口名 . 属性名
```

虽然这种写法会使 DOM 的书写变得冗长，但以这样的方式书写 JavaScript 代码就可以清楚地知道正

在对哪一个接口的对象进行操作，将有助于增进对代码的理解。所以还是应采用这种方法。同时，在方法名之后接上 () 之后，就能够清楚地知道该元素是一个方法。

9.2 DOM 的基础

9.2.1　标签、元素、节点

在对 HTML 以及 DOM 进行讨论时，我们常常会混用标签、元素、节点等术语。故在此再次对它们的定义进行确认。

■ 标签

标签是一种用于标记的字符串，其作用为对文档的结构进行指定。通常都会有起始标签与结束标签。在结束标签中，有一些可以被省略，如 <p> 标签。同时也有一些标签不存在结束标签，例如 <input> 标签（代码清单 9.1）。说到底标签只是用于书写场合，在谈论 DOM 的话题时几乎不会使用。

代码清单 9.1　标签

```
<div><!-- div 的起始标签 -->
    <p> <!-- p 的结束标签可以省略 -->
    <input type="button"> <!-- input 只有起始标签而没有结束标签 -->
</div><!-- div 的结束标签 -->
```

■ 元素、节点

比较容易产生混淆的是元素和节点的概念。元素和节点之间略有一些继承关系，其中节点是父类概念。节点具有 nodeType 这一属性，如果其值为 ELEMENT_NODE(1)，该节点则是一个元素。

表 9.5 总结了 HTML 文档常用的节点[①]。

表 9.5　在 HTML 文档中使用的节点

节点	节点类型常量	节点类型的值	接口
元素节点	ELEMENT_NODE	1	Element
属性节点	ATTRIBUTE_NODE	2	Attr
文本节点	TEXT_NODE	3	Text
注释节点	COMMENT_NODE	8	Comment
文档节点	DOCUMENT_NODE	9	Document

9.2.2　DOM 操作

JavaScript 的作用是使网页能够执行某些功能。为了实现这些功能，必须对 DOM 进行操作。通过选择某个 DOM 元素并改写其属性，或创建一个新的 DOM 元素，就能够给予用户视觉反馈，以实现交互功能。之后，将从选择（9.3 节点的选择）、创建（9.4 节点的创建与新增）、更改（9.5 节点的内容变更）与删除（9.6 节点的删除）四个方面对 DOM 的相关操作进行说明。

9.2.3　Document 对象

Document 对象是 DOM 树结构中的根节点。虽然这是一个根节点，在 HTML 文档中却不用书写其对应的标签。例如，虽然 <html> 标签与 <body> 标签分别对应 Document 对象中的 documentElement 属性与 body 属性，但却没有与 Document 对象自身相对应的标签。这是因为 Document 对象是一种用于表示整个

① 在本章中，元素一词所指代的对象仅指元素节点。对于不局限于元素节点的情况，将使用节点一词来指代广义的节点。

HTML 文档的对象。

可以通过 JavaScript 中的 document 这一全局变量来访问 Document 对象。准确地说，document 是 window 对象中的一个属性。不过，由于 window 对象是一个全局对象，因此在对其属性进行访问时可以将 window. 省略不写。

实际上，在通过 JavaScript 表示 HTML 文档时，所有的全局变量都是 window 对象的属性。可以通过下面的代码对此进行确认。

```
var global_variable = 'Global Variable';
alert(window.global_variable) === global_variable);  // => true
```

顺便提一下，window 对象并没有包含于 DOM 树结构之中。如前面所讲，Document 对象在 DOM 树结构中是根节点，因此也无法通过下面将要介绍的方法来取得 Document 对象的父节点。

9.3 节点的选择

9.3.1 通过 ID 检索

在 JavaScript 中，如果要对 HTML 文档中的指定节点进行选择，Document.getElementById() 方法是一种最为常见的手段。该方法可以像下面这样书写。

```
var element = document.getElementById('foo');
```

这样就能够取得 ID 为 foo 的元素。ID 在 DOM 树中必须是唯一的。在 DOM 中并没有对存在多个相同 ID 的情况做出规定。不过，大部分的浏览器都采用了返回第一个找到的元素的方式。

不过即便如此，也不应该根据这一规则来进行设计。这样的做法是错误的，ID 必须是唯一的（代码清单 9.2）。

代码清单 9.2　在同时存在多个相同 ID 时 getElementById() 方法的行为

```
<div id="foo">first</div>
<div id="foo">second</div>
<script>
    var element = document.getElementById('foo');
    alert(element.innerHTML);  // => 大部分浏览器都会返回 first。不过这并不是一种绝对标准
</script>
```

9.3.2 通过标签名检索

可以通过下面这样的方式，使用 Element.getElementsByTagName() 方法来取得具有该标签名的所有节点。标签名还可以使用 '*' 作为通配符。可以通过 '*' 来获取所有元素。

```
var spanElements = document.getElementsByTagName('span');     // 仅获取 span 元素
var allElements = document.getElementsByTagName('*');         // 获取所有的元素
```

Document.getElementById() 是只存在于 Document 对象中的方法，而 Element.getElementsByTagName() 则是同时存在于 Document 对象与 Element 对象这两者中的方法。在执行某个 Element 对象的 getElementsByTagName() 方法时，该 Element 对象的子孙节点中具有指定标签名的元素也将被获取（代码清单 9.3）。

代码清单 9.3　getElementById() 与 getElementsByTagName()

```
<body>
<p id='foo'>
    <span>a</span>
```

```
    <span>b</span>
    <span>c</span>
</p>
<p id='bar'>
    <span>x</span>
</p>
<script>
    var foo = document.getElementById('foo');
    // 在 Element 对象中没有 getElementById() 方法
    alert(foo.getElementById)  // => undefined
    // 在 Element 对象中存在 getElementsByTagName() 方法
    alert(foo.getElementsByTagName)    // => function getElementsByTagName() { [native code] }
    // 从 foo 的子孙节点中取得元素 span
    var fooSpans = foo.getElementsByTagName('span');
    alert(fooSpans.length);    // => 3
    // 从整个文档中获取元素 span
    var allSpans = document.getElementsByTagName('span');
    alert(allSpans.length);    // => 4
</script>
</body>
```

■ Live 对象的特征

在这里需要注意的是，getElementsByTagName() 所能取得的对象是一个 NodeList 对象，而不是单纯的 Node 对象的数组。而 NodeList 对象的一大特征就是它是一个 Live 对象。可以写出代码清单 9.4 这样的代码。

代码清单 9.4　Live 对象

```
<div id="foo">
    <span>first</span>
    <span>second</span>
</div>
<script>
    var elems = document.getElementsByTagName('span');
    alert(elems.length);        // => 2
    var newSpan = document.createElement('span');
    newSpan.appendChild(document.createTextNode('third'));
    var foo = document.getElementById('foo');
    foo.appendChild(newSpan);
    alert(elems.length);        // => 3
</script>
```

在上面的代码中，最初取得的 elems.length 值为 2，这是很显然的。之后，通过 JavaScript 新增了一个 span 元素。再一次显示 elems.length 时其值变为了 3。

如果是在新增了 span 之后再执行 getElementsByTagName()，自然不会觉得有什么问题，但这里明明是一个在新增 span 之前就已经取得的 NodeList 对象，却能够知道新增了 span 之后的状态，是不是觉得有点奇怪？而这就是 Live 对象的一个特征。

Live 对象始终具有 DOM 树实体的引用。因此，对 DOM 树做出的变更也会在 Live 对象中得到体现。

■ 在对 Live 对象进行操作时的注意点

在使用 Live 对象时，必须对代码清单 9.5 中这样的 for 循环多加注意。

代码清单 9.5　Live 对象中存在的陷阱

```
<div>sample text</div>
<script>
    var divs = document.getElementsByTagName('div');
    var newDiv;
    for (var i = 0; i < divs.length; i++)
        newDiv = document.createElement('div');
        newDiv.appendChild(document.createTextNode('new div'));
        divs[i].appendChild(newDiv);
```

```
    }
</script>
```

在上面的代码中，将会首先取得所有的 div 元素，然后在 for 循环中创建新的 div 元素并添加至这些 div 元素中。于是，作为循环条件的 divs.length 的值将会不断地增加 1，而无法离开循环。对于这种情况，只要在开始时对 divs.length 进行求值，就可以避免无限循环（代码清单 9.6）。

代码清单 9.6 避免进入 Live 对象中的陷阱

```
<div>sample text</div>
<script>
    var divs = document.getElementsByTagName('div');
    var newDiv;
    // 先取得 divs.length 的值，并以此作为循环的条件
    for (var i = 0, len = divs.length; i < len; i++)
        newDiv = document.createElement('div');
        newDiv.appendChild(document.createTextNode('new div'));
        divs[i].appendChild(newDiv);
    }
</script>
```

■ Live 对象的性能

Live 对象是否方便取决于具体的使用场景，不过如果只对性能进行讨论的话，它的性能确实是较差的。与直接使用 getElementsByTagName() 的结果的情况相比，先将该结果转换为 Array 之后再进行使用性能更好。

可以通过 Array.prototype.slice() 方法将一个 NodeList 对象转换为一个 Array（代码清单 9.7）。

代码清单 9.7 getElementsByTagName() 的返回值

```
var nodeList = document.getElementsByTagName('span');
alert(nodeList instanceof NodeList);                  // => true
alert(nodeList instanceof Array);                     // => false
var array = Array.prototype.slice.call(nodeList);     // 将 NodeList 对象转换为 Array 对象
alert(nodeList instanceof NodeList);                  // => false
alert(nodeList instanceof Array);                     // => true
```

不过，这种方法不适用于 Internet Explorer 8 以及更早的版本。如果使用 Array.prototype.slice() 方法来处理 Element.getElementsByTagName() 就会发生错误。因此，如果要将其转换为 Array，就不得不对一个个元素进行设置。进一步来讲，在 Internet Explorer 8 以及更早的版本中，Element.getElementByTagName() 所获取的并不是 NodeList 对象，而是一个 HTMLCollection 对象。这与 DOM Level 1 的定义是不一致的。Element.getElementsByTagName() 方法应该返回一个 NodeList 才对。顺便提一下，HTMLCollection 也是一种 Live 对象，在这一点上它与 NodeList 是相同的（代码清单 9.8）。

代码清单 9.8 Internet Explorer 中的 getElementsByTagName()

```
// 在 Internet Explorer 8 以及更早的版本中运行
var htmlCollection = document.getElementsByTagName('span');
alert(htmlCollection instanceof HTMLCollection);     // => true
alert(htmlCollection instanceof NodeList);           // => false
alert(htmlCollection instanceof Array);              // => false
for (var i = 0, len = htmlCollection.length; i < len; i++) {
    array[i] = htmlCollection[i];
}
```

可以通过执行代码清单 9.9 中这样的代码对 NodeList 的性能进行确认。

代码清单 9.9 根据 NodeList 的操作方式不同，其性能也会有所差异

```
<div>
<!-- 写有 1000 个 <span> 标签 -->
</div>
```

```
<script>
    var elems, len;
    // 直接使用 NodeList，且每次获取其 length
    console.time('直接使用 NodeList，且每次获取其 length');
    elems = document.getElementsByTagName('span');
    for (var i = 0; i < 1000; i++) {
        for (var j = 0; j < elems.length; j++) {
            elems[j];
        }
    }
    console.timeEnd('直接使用 NodeList，且每次获取其 length');
    // 直接使用 NodeList
    console.time('直接使用 NodeList');
    elems = document.getElementsByTagName('span');
    len = elems.length;
    for (var i = 0; i < 1000; i++) {
        for (var j = 0; j < len; j++) {
            elems[j];
        }
    }
    console.timeEnd('直接使用 NodeList');
    // 将其转换为 Array 后使用，且每次获取其 length
    console.time('将其转换为 Array 后使用，且每次获取其 length');
    // 在 Internet Explorer 8 以及更早版本中将会产生错误
    elems = Array.prototype.slice.call(document.getElementsByTagName('span'));
    for (var i = 0; i < 1000; i++) {
        for (var j = 0; j < elems.length; j++) {
            elems[j];
        }
    }
    console.timeEnd('将其转换为 Array 后使用，且每次获取其 length');
    // 将其转换为 Array 后使用
    console.time('将其转换为 Array 后使用 ');
    // 在 Internet Explorer 8 以及更早版本中将会产生错误
    elems = Array.prototype.slice.call(document.getElementsByTagName('span'));
    len = elems.length;
    for (var i = 0; i < 1000; i++) {
        for (var j = 0; j < len; j++) {
            elems[j];
        }
    }
    console.timeEnd('将其转换为 Array 后使用 ');
</script>
```

运行结果如下所示。可以看到，通过 NodeList 对象获取元素将会花费相当的时间，同时对 length 属性进行引用也会产生一些时间上的开销。

通过这段代码不难理解，要在 for 循环中使用 NodeList 时，先将其转换为 Array 对象后再使用比较好。

```
直接使用 NodeList，且每次获取其 length: 276ms
直接使用 NodeList: 155ms
将其转换为 Array 后使用，且每次获取其 length: 22ms
将其转换为 Array 后使用 : 20ms
```

除了 NodeList 外，还有其他的 Live 对象。前面提到过，HTMLCollection 也是一个 Live 对象。在 Internet Explorer 8 以及更早的版本中，getElementsByTagName() 将会返回一个 HTMLCollection 对象。此外，用于容纳文档中的表单对象的 document.forms 等也都是 HTMLCollection 对象。表 9.6 总结了 DOM Level 1 中定义的 HTMLCollection 对象。

表 9.6　在 DOM Level 1 中定义的 HTMLCollection

HTMLCollection	说明
document.images	文档中所有的 img 元素
document.applets	文档中所有的 Java Applet 对象
document.links	文档中所有的链接元素（那些被指定了 href 属性的元素）
document.forms	文档中所有的 form 元素

（续）

HTMLCollection	说明
document.anchors	文档中所有的锚元素（那些被指定了 name 属性的元素）
form.elements	表单中所有的 input 元素
map.areas	图像映射中所有的 area 元素
table.rows	表格中所有的 tr 元素
table.tBodies	表格中所有的 tbody 元素
tableSection.rows	表格区段（thead 元素、tfoot 元素）中所有的 tr 元素
row.cells	表格的一行中所有的 td 元素与 th 元素

由于 Live 对象的这一性能问题，通常都会将 Live 对象先转换为 Array，之后再作为结果返回。这一部分的操作通常会很好地隐藏起来，因此并不需要做什么额外处理。如果运行结果和预想的情况不同，则应该尝试确认所返回的对象是否是 Live 对象。这一步骤随处存在潜在的陷阱。

虽然在编写 JavaScript 程序的过程中不需要过分在意变量的数据类型，但在实际的开发中也不要忘了“对象毕竟还是具有某种数据类型的”这一事实。

9.3.3 通过名称检索

通过 HTMLDocument.getElementsByName() 方法，可以将 name 属性的值作为限定条件来获取属性。不过因为只能在 form 标签或 input 标签等标签中使用 name 属性，所以与 getElementById() 相比，它的使用频率较低。

9.3.4 通过类名检索

通过使用 HTMLElement.getElementsByClassName() 方法，就可以获取指定类名的元素（代码清单 9.10）。其中的类名可以指定多个值。如果想要指定多个类名，则需要使用空白符作为分隔字符串。也就是类似于 'classA classB' 的形式。这时，会取得 classA 与 classB 这两个类名所指定的元素。

该方法并不属于 DOM Core 或 DOM HTML 标准，而是一种 HTML5 规定的功能。不过在实际使用中并不需要对此过于在意。

从这是一个在 HTML5 标准中定义的方法这一点上也能知道，只有现代的浏览器才对该方法提供了支持。在 Internet Explorer 8 以及更早的版本中无法使用这一方法。不过其实在很多 JavaScript 库中，都有类似于 getElementsByClassName() 这一方法的实现，所以只要利用库的话，同样能够简单地在 Internet Explorer 8 以及更早的版本中使用这样的功能。

代码清单 9.10　getElementsByClassName() 方法

```
<body>
<p id='foo'>
    <span class='matched'>a</span>
    <span class='matched unmatched'>b</span>
    <span class='unmatched'>c</span>
</p>
<p id='bar'>
    <span class='matched'>x</span>
</p>
<script>
    var foo = document.getElementById('foo');
    // 从 foo 的子孙节点中获取被指定为 matched 类的元素
    var fooMatched = foo.getElementsByClassName('matched');
    alert(fooMatched.length);      // => 2
    // 如果要指定多个类名，则需要通过空白符分隔
    alert(foo.getElementsByClassName('matched unmatched').length);// => 1
    // 在指定了多个类名的时候，改变类名间的顺序也不会有什么影响
    alert(foo.getElementsByClassName('unmatched matched').length);// => 1
    // 从整个文档中获取类名为 matched 的元素
```

```
    var allMatched = document.getElementsByClassName('matched');
    alert(allMatched.length);      // => 3
</script>
</body>
```

9.3.5　父节点、子节点、兄弟节点

接下来将说明如何获取某个节点的父节点、子节点以及兄弟节点。在一个节点中包含了一些用于引用其他节点的属性（表 9.7）。

表 9.7　一些用于引用相关节点的属性

属性名	能够获取的节点
parentNode	父节点
childNodes	子节点列表
firstChild	第一个子节点
lastChild	最后一个子节点
nextSibling	下一个兄弟节点
previousSibling	上一个兄弟节点

此外，空白符也会作为文本节点处理。换行符也包含在这种情况之中。在书写 HTML 的过程中为了确保可读性，通常会在标签之间加入一些换行符。然而，这样一来在换行处就会存在一些空白节点。于是，在使用 firstChild 的时候就会首先取得这些空白节点。在通过 firstChild 等属性引用元素时，请务必对此加以注意（代码清单 9.11）。

代码清单 9.11　用于引用相关节点的属性的具体使用示例

```
<body>
    <div id="a">
        <div id="b"></div>
        <div id="my">
            <div id="c">
                <div id="d"></div>
            </div>
            <div id="e"></div>
        </div>
        <div id="f"></div><div id="g"></div>
    </div>
    <script>
        var my = document.getElementById('my');
        var elem;
        elem = my.parentNode;
        alert(elem.id);                 // => 'a'
        // 子元素
        elem = my.firstChild;
        alert(elem.id);                 // => undefined       // 取得的是空白节点
        elem = my.nextSibling;
        alert(elem.id);                 // => 'c'
        elem = my.lastChild;
        alert(elem.id);                 // => 'e'
        var children = my.childNodes;
        alert(children[0].id);          // => undefined       // 取得的是空白节点
        alert(children[1].id);          // => 'c'
        alert(children[2].id);          // => undefined       // 取得的是空白节点
        alert(children[3].id);          // => 'e'
        // 兄弟元素
        elem = my.previousSibling.previousSibling;
        alert(elem.id);                 // => 'b'
        elem = my.nextSibling.nextSibling;
        alert(elem.id);                 // => 'f'
        elem = elem.nextSibling;
        alert(elem.id);                 // => 'g'
        // 由于在 div#f 与 div#g 之间没有空白符或换行符，因此 div#f 的 nextSibling 是 div#g
```

```
    </script>
    </body>
```

在这里需要注意的是，通过 childNodes 获取的对象都是 NodeList 对象。而 NodeList 对象是一种 Live 对象。前面提到过，Live 对象的性能并不太好，如果在 childNodes 可能会很大时，最好将其先转换为 Array 之后再使用。

总之，由于上面所介绍的 firstChild 等属性也会包含空白节点，因此最终取得的节点并不一定就是直观上所以为的那个。如果在取得了节点之后要修改节点，则还须判断节点是否为空白节点，很不方便。因此，另外还制定了一些用于获取不包含空白节点与注释节点的元素的 API，它们在一些浏览器中已经实现（表 9.8）。

表 9.8　一些用于引用相关元素的属性

属性名	能够获取的元素
children	子元素节点列表
firstElementChild	第一个子元素
lastElementChild	最后一个子元素
nextElementSibling	下一个元素
previousElementSibling	上一个元素
childElementCount	子元素的数量

除了 children 之外，其他的几个被称为 Element Traversal API（元素遍历 API）。尽管 children 不属于 Element Traversal API，不过所有主流的浏览器都对其进行了实现，所以可以将它作为一种获取子元素的方法。虽然 Internet Explorer 也支持 children，不过 Internet Explorer 中的 children 将会返回一个包含空白节点的 NodeList，对此请加以注意。Internet Explorer 还真是一个麻烦的浏览器。

如果用 Traversal API 改写之前的例子，则能够得到代码清单 9.12 这样的代码。可以看到，由于忽略了空白节点，现在的代码比使用 firstChild 等属性时要更为直观。

代码清单 9.12　用于引用相关元素的属性的具体使用示例

```
<body>
    <div id="a">
        <div id="b"></div>
        <div id="my-id">
            <div id="c">
                <div id="d"></div>
            </div>
            <div id="e"></div>
        </div>
        <div id="f"></div><div id="g"></div>
    </div>
    <script>
        var my = document.getElementById('my-id');
        var elem;
        elem = my.parentNode;
        alert(elem.id);                // => 'a'
        // 子元素
        elem = my.firstElementChild;
        alert(elem.id);                // => 'c'
        elem = my.lastElementChild;
        alert(elem.id);                // => 'e'
        var children = my.children;
        alert(children[0].id);         // => 'c'
        alert(children[1].id);         // => 'e'

        // 兄弟元素
        elem = my.previousElementSibling;
        alert(elem.id);                // => 'b'
        elem = my.nextElementSibling;
        alert(elem.id);                // => 'f'
        elem = elem.nextElementSibling;
```

```
        alert(elem.id);                 // => 'g'
    </script>
</body>
```

9.3.6 XPath

尽管通过上述的 getElementById() 与 childNodes 等方法就能够访问所有节点，不过这还称不上是构成了完整的节点指定方法。通过 XPath 来指定节点是一种更为灵活的方式。

通过 XPath 方式，能够很容易地实现复杂的节点指定操作。例如，指定一个 id 为 main 的 div 元素中的第三个以 content 为 class 值的 p 元素中的某个 a 元素，且该元素的 href 值的起始部分为 http://example.com/（代码清单 9.13）。

代码清单 9.13 XPath 对象的 HTML 结构

```
<div id="main">
    <p class="content">
        <a class="link" href="http://example.com"/>1st link</a>
    </p>
    <p class="dummy"></p>
    <p class="content">
        <a href="http://example.com/">2nd link</a>
    </p>
    <p class="content">
        <a href="http://foobar.example.com/">3rd link</a>
        <a href="http://example.com/">4th link</a>
    </p>
    <a href="http://example.com/">5th link</a>
</div>
```

对于代码清单 9.13 中的 HTML 代码，以前面所说的条件应该是会取得表示 4th link 的 a 元素。如果要通过 XPath 来实现这一操作，则需要使用代码清单 9.14 中的方式。

代码清单 9.14 XPath 的使用示例

```
<script>
var result = document.evaluate(
    // id 为 main 的 div / 包含了值为 content 的 class 的 p 元素中的第三个 /href 值的起始部分
为 http://example.com/ 的 a 元素
    '//div[@id="main"]/p[contains(@class, "content")][3]/a[starts-with(@href,
"http://example.com/")]',
    document,
    null,
    XPathResult.ORDERED_NODE_SNAPSHOT_TYPE,
    null
);
alert(result.snapshotLength);       // => 1
var elem = result.snapshotItem(0);
alert(elem.innerHTML);              // => 4th link
</script>
```

通过这样的方式使用 XPath，就能够灵活地指定并取得某个特定的 DOM 元素。Document.evaluate() 方法将会接收 5 个参数，可能会有些不容易理解，不过在熟悉之后就会发现它也是很简单的。

它的第 1 个参数是用于求值的 XPath 表达式的字符串。

它的第 2 个参数用于指定文档中的节点。XPath 表达式将会在被指定的这一节点中进行求值匹配并返回结果。如果将参数指定为 document，则会搜索整个文档。如果能够确定搜索的范围，最好在此指定一个恰当的值。这样就不会进行无意义的搜索，从而有望提高执行性能。

在第 3 个参数中，可以指定用于返回 URI 命名空间的函数。这一参数是用于需要使用命名空间前缀的 XML 文档之中，在 HTML 文档中不需要使用。传递 null 值即可。

第 4 个参数指定了返回求值结果时所要使用的对象类型。evaluate() 方法的返回值为一个 XPathResult 对象，而 XPathResult 对象又含有多种类型。该参数正是用于指定其具体的类型。能够用于指定的值为 0 至 9 的数字。所有的这些都在 XPathResult 接口中被定义为了常量。

表 9.9 总结了该参数被指定为不同的值时将会返回的对象。

表 9.9　evaluate 方法的第 4 个参数的值与其返回值之间的关系

常量	值	所返回的对象
ANY_TYPE	0	一个包含了与求值结果相符类型的结果的集合。如果该结果是一个节点的集合，则返回的对象与被指定为 UNORDERED_NODE_ITERATOR_TYPE 时的相同
NUMBER_TYPE	1	数值
STRING_TYPE	2	字符串
BOOLEAN_TYPE	3	真假值
UNORDERED_NODE_ITERATOR_TYPE	4	一个节点集合的迭代器。不限定顺序
ORDERED_NODE_ITERATOR_TYPE	5	一个节点集合的迭代器。顺序与其在文档中出现的顺序相一致
UNORDERED_NODE_SNAPSHOT_TYPE	6	一个节点集合的快照。不限定顺序
ORDERED_NODE_SNAPSHOT_TYPE	7	一个节点集合的快照。顺序与其在文档中出现的顺序相一致
ANY_UNORDERED_NODE_TYPE	8	与表达式相匹配的节点中的任意一个。并不一定是第一个与表达式匹配的节点
FIRST_ORDERED_NODE_TYPE	9	在文档内第一个与表达式匹配的节点

返回值为迭代器与返回值为快照的区别在于，它们对在 evaluate() 方法被执行之后文档中产生的变更的处理方式不同。对于迭代器来说，如果在取得结果之后对文档进行了修改，则会在执行 iterateNext() 方法时抛出异常。而对于快照来说，则不会抛出异常。不过这时也仅仅是会在执行 evaluate() 方式所取得结果的基础上继续处理（代码清单 9.15）。

代码清单 9.15　迭代器与快照的区别

```
// 获取迭代器
var iterator = document.evaluate(
    '//div[@id="main"]/p',
    document,
    null,
    XPathResult.ORDERED_NODE_ITERATOR_TYPE,
    null
);

// 在取得了迭代器之后，继续以一定的条件向文档中增加新的节点
var newParagraph = document.createElement('p');
document.getElementById('main').appendChild(newParagraph);
newParagraph.appendChild(document.createTextNode('This is a new paragraph.'));
try {
    node = iterator.iterateNext(); // 将会抛出 INVALID_STATE_ERR 异常
} catch (e) {
    console.log(e);
}

// 获取快照
var snapshot = document.evaluate(
    '//div[@id="main"]/p',
    document,
    null,
    XPathResult.ORDERED_NODE_SNAPSHOT_TYPE,
    null
);

// 在取得了迭代器之后，继续以一定的条件向文档中增加新的节点
var anotherParagraph = document.createElement('p');
document.getElementById('main').appendChild(anotherParagraph);
newParagraph.appendChild(document.createTextNode('This is another paragraph.'));
for (var i = 0; i < snapshot.snapshotLength; i++) {
```

```
    console.log(snapshot.snapshotItem(i) === anotherParagraph);
    // 所有的结果都将为 false
    // 也就是说，anotherParagraph 并没有包含在 snapshot 中
    // 但也不会抛出异常
}
```

这个参数常常会被指定为 ORDERED_NODE_SNAPSHOT_TYPE 值。

最后的第 5 个参数用于指定一个已有的 XPathResult 对象。如果指定了该参数，则能够重复使用 XPathResult 对象。如果没有指定该参数，则会新生成一个 XPathResult 对象。通常只要将这一参数指定为 null 就不会有什么问题了。

在使用 XPath 的过程中需要注意的是，Internet Explorer 不支持这一功能。即使是 Internet Explorer 9 也无法使用。不过，已经有了一些用于为 Internet Explorer 添加对 XPath 的支持的库（JavaScript-XPath），只要使用这些库就能够解决这一问题。

可以从下面的 URL 下载 JavaScript-XPath。

http://coderepos.org/share/wiki/JavaScript-XPath

9.3.7　Selector API

尽管通过 XPath 就能够灵活地指定并取得所需元素，但其使用方法多少有些复杂。而 Selector API 正是一种比 XPath 更为简单，而同时又保持了相当灵活性的元素获取方式。

通过 Selectors API 对元素指定的方式与其在 CSS 中指定元素的方式相同。所以，它可以简单地实现与 getElementById() 或 getElementsByTagName() 相同的操作（代码清单 9.16）。而通过 querySelectorAll() 则可以获取所有符合条件的元素。此时，如果有多个元素同时符合条件，querySelector() 将只会返回第一个与条件相符的元素。

代码清单 9.16　Selectors API 的使用示例

```
var a = document.querySelector('#foo');
var b = document.getElementById('foo');
alert(a === b);        // => true
var c = document.querySelectorAll('div');
var d = document.getElementsByTagName('div');
alert(c[0] === d[0]);      // => true
```

此外，还可以像代码清单 9.17 这样，通过 Selector API 实现与使用 XPath 时相同的功能，以取得和代码清单 9.14 相同的结果。对于熟悉 CSS 选择器的人来说，这种方式应该会比 XPath 具有更好的可读性。

代码清单 9.17　Selector API 的使用示例 2

```
<script>
    var elem = document.querySelector(
        'div#main > p.content:nth-of-type(4) > a[href^="http://example.com/"]');
    alert(elem.innerHTML);     // => 4th link
</script>
```

这里有一个需要注意的要点。querySelectorAll() 方法所返回的对象不同于通过 getElementsByTagName() 或 childNodes 等方式所取得的 NodeList 对象。通过 querySelectorAll() 所取得的是一个 StaticNodeList 对象。

NodeList 与 StaticNodeList 的区别在于，更改对象之后是否会将该更改反映于 HTML 文档之中。如果对 NodeList 对象进行了更改，则在 HTML 文档中也会体现出相应的变化。然而，如果更改了 StaticNodeList 对象，在 HTML 文档中并不会反映出这一更改。如果没有意识到这一区别的话，会发现对 StaticNodeList 进行操作的结果与对 NodeList 进行操作的结果是不同的，程序并没有按照期望的方式运行。

9.4 节点的创建与新增

可以通过 Document.createElement() 方法或 Document.createTextNode() 方法来创建节点。此外还有一种不太常用的方法，即还可以通过 Document.createComment() 方法来创建一个注释。

如果仅仅是创建了一个节点，对于 HTML 文档来说并不会发生什么变化。只有将创建的节点加入 DOM 树之后该节点才会在浏览器中显示。

如果要将节点新增为某一节点的最后一个子元素，则可以使用 Node.appendChild() 方法。而如果要将节点插入至某一元素所在的位置，则需要使用 Node.insertBefore() 方法（代码清单 9.18）。

代码清单 9.18　节点的创建与新增

```
var elem = document.createElement('div');         // 创建一个 div 元素
var text = document.createTextNode('This is a new div element.'); // 创建一个文本节点
document.body.appendChild(elem);                  // 将所创建的 div 元素添加至 body 之下
elem.appendChile(text);                           // 将文本节点添加至所创建的 div 元素中
var comment = document.createComment('this is comment'); // 创建一个注释节点
document.body.insertBefore(comment, elem);        // 在 elem 之前插入该注释节点
```

9.5 节点的内容更改

如果改写所取得节点的属性，则这一改动也会在 HTML 文档中得到体现。另外还可以通过 Node.replaceChild() 方法替换节点（代码清单 9.19）。

代码清单 9.19　节点的替换

```
var newNode = document.createElement('div');
var oldNode = document.getElementById('foo');
var parentNode = oldNode.parentNode;
parentNode.replaceChild(newNode, oldNode);
```

9.6 节点的删除

可以通过 Node.removeChild() 方法来删除节点（代码清单 9.20）。

代码清单 9.20　节点的删除

```
var elem = document.getElementById('foo');
elem.parentNode.removeChild(elem);
```

9.7 innerHTML/textContent

9.7.1 innerHTML

通过上面介绍的这些方法，我们已经能够自由修改 HTML 文档了。不过，需要修改大量元素时，如果逐一使用 createElement() 或 appendChild()，则会显得过于冗长。其实还有一种更为简单的书写方式，即使用 HTMLElement 的 innerHTML 属性。

在对 innerHTML 属性赋值之后，浏览器将会分析其内容，并将分析结果设为该元素的子元素（代码清单 9.21）。

不过，innerHTML 属性并不是一种在 DOM 标准中被定义的功能，而是一种在 HTML5 中被定义的属性。但因为 Internet Explorer 在很早之前就实现了这一功能，所以在大部分的浏览器中都能使用 innerHTML。

代码清单 9.21　innerHTML 的使用示例

```
var elem = document.getElementById('foo');
elem.innerHTML = '<div>This is a new div element.</div>';
```

9.7.2　textContent

innerHTML 属性可以以 HTML 字符串的形式被引用，而 textContent 属性则可以取得包含子元素在内的纯文本部分（代码清单 9.22）。因此，如果设定 textContent 属性，就能够将子元素全部删除，并将其替换为一个文本节点。

Internet Explorer 有一个 innerText 属性可以实现与其相同的功能。textContent 属性是一种在 DOM Level 3 Core 中被定义的属性。

代码清单 9.22　textContent 的使用示例

```
var elem = document.getElementById('foo');
elem.textContent = '<div>Is this a new div element?</div>';
// => 不会创建 div 元素。在浏览器中将会直接显示该字符串
```

9.8　DOM 操作的性能

在客户端 JavaScript 中，DOM 操作是不可或缺的。通过 DOM 操作可以实现内容的改写，而如果在显示上发生了变化，浏览器自然要重新绘制画面。而画面的重新绘制这一步骤是需要花费开销的，所以应当尽可能避免重新绘制画面。

下面对向画面中新增 10 个 div 元素的情况进行讨论。如果只是简单地像代码清单 9.23 这样书写的话，则会执行 10 次画面刷新。

代码清单 9.23　性能较差的书写方式

```
var parent = document.getElementById('parent');
for (var i = 0; i < 10; i++) {
    var child = document.createElement('div');
    // 向父元素中添加子元素。在添加时将会重新绘制画面
    parent.appendChild(child);
}
```

与之相对，如果按照代码清单 9.24 中这样使用 DocumentFragment，则可以将画面的重绘次数降低至 1 次。

代码清单 9.24　使用 DocumentFragment

```
var fragment = document.createDocumentFragment();
for (var i = 0; i < 10; i++) {
    var child = document.createElement('div');
    // 向 DocumentFragment 添加子元素
    fragment.appendChild(child);
}

// 向父元素中添加 DocumentFragment
// 虽然添加的是 DocumentFragment，但实际上添加的仅仅是 DocumentFragment 的子元素
document.getElementById('parent').appendChild(fragment);
```

像这样先修改 DocumentFragment，最后再对实际的 document 对象进行操作的话，就可以避免不必要的画面重绘。

第 10 章 事件

本章讲解事件的处理，这是客户端 JavaScript 中最为重要的概念之一。为了使用 JavaScript 实现各种各样的功能，必须恰当地处理不同的事件。

10.1 事件驱动程序设计

在 JavaScript 中，最为重要的一件事就是对事件进行处理。与通常的 GUI 应用程序相同，Web 应用程序也是通过事件驱动程序设计的方式来实现其功能的。在事件驱动程序设计中，需要注册不同事件的处理方式。

在注册了事件的处理方式之后，浏览器就会在该事件发生时执行所注册的处理方式。所登录的处理方式被称作事件处理程序、事件句柄或事件侦听器。

对于 Web 应用来说，有下面这些代表性的事件：点击某个元素、将鼠标移动至某个元素上方、按下键盘上某个键，等等。此外，读取页面或跳转至其他页面等行为也会引发事件。根据这些不同的用户操作，浏览器会触发相应的事件。之后，执行事件处理程序，处理被触发的事件。

所以说，JavaScript 程序设计的基本内容之一就是获取需要对事件进行捕捉的元素，并针对该元素注册相应的事件处理程序。

DOM Level 2 中定义了标准的事件模型。大部分现代浏览器都是根据这一标准实现的。但是，在 Internet Explorer 8 以及更早版本中，采用了自定义的事件模型实现方式。从功能上来说，这确实和标准事件模型没有太大的差别，但 API 是完全不同的，应当加以注意。这里会以标准的事件模型为基础进行说明。对于 Internet Explorer 相关的说明，请参见与跨浏览器支持相关的内容。

10.2 事件处理程序 / 事件侦听器的设定

对事件的处理方式被称为事件处理程序或事件侦听器，但这两者之间其实是有区别的。它们的设定方法并不相同，因此，两者支持的处理元素数量也不同。对于 1 个元素或事件，只能设定 1 个事件处理程序。而与之相对的，可以为其同时设定多个事件侦听器。

下面是一些对事件处理进行设定的方式。

- 指定为 HTML 元素的属性（事件处理程序）
- 指定为 DOM 元素的属性（事件处理程序）
- 通过 EventTarget.addEventListener() 进行值定（事件侦听器）

接下来，将会对各种方式进行详细说明。

10.2.1　指定为 HTML 元素的属性

将事件处理程序指定为 HTML 元素的属性是一种最为简单的设定事件处理程序的方式。在下面的例子中，将会在发生按钮点击事件时显示含有 bar 和 baz 消息的提示对话框。

```
<input id="foo" type="button" value="foo" onclick="alert('bar');alert('baz')">
```

在这个例子中，通过字符串对 onclick 事件处理程序将要执行的 JavaScript 代码进行了设定。如果代码包含多行，则可以通过分号分隔。当然，事先另外定义一个函数之后再执行该函数的方式也不会有问题。

这种方式的优点在于，设定步骤非常简单，并且能够确保事件处理程序会在载入时被设定。而如果使用之后所介绍的一些方法则可能会产生一些问题。即元素在被载入时，其事件处理程序可能还没有被注册，这时用户执行任何本应触发事件的操作，也不会有任何效果。与之相对地，将事件处理程序指定为 HTML 元素的属性的话，就能够确保它在载入的同时被设定。

在书写上也有一些需要注意的地方，就是这里的 onclick 全都是以小写字母书写的。HTML 不会区分大小写字母，所以即使在这里使用了 onClick 也不会有什么差别。但是，XHTML 则会区分大小写字母。考虑到这点，最好还是使用全部小写的 onclick，这样将有助于提高代码的兼容性。

表 10.1 是事件处理程序的一览表。

表 10.1　事件处理程序

事件处理程序名	触发的时机
onclick	鼠标点击操作
ondblclick	鼠标双击操作
onmousedown	按下了鼠标按键
onmouseup	放开了鼠标按键
onmousemove	鼠标指针在元素上方移动
onmouseout	鼠标指针从元素上方离开
onmouseover	鼠标指针移动至了元素上方
onkeydown	按下了键盘按键
onkeypress	按过了键盘按键
onkeyup	放开了键盘按键
onchange	更改了 input 元素的内容
onblur	input 元素失去了焦点
onfocus	input 元素获得了焦点
onselect	文本被选取
onsubmit	按下了表单的提交按钮
onreset	按下了表单的重置按钮
onload	载入完成
onunload	文档的载入被撤销（例如页面跳转等情况时）
onabort	图像的读取被中断
onerror	图像读取过程中发生错误
onresize	窗口尺寸发生改变

如果事件处理程序返回了一个 false 值，则会取消该事件的默认行为。例如，当 onsubmit 事件处理程序返回了一个 false 时，表单的内容将不会被发送。这在使用 onsubmit 事件处理程序验证表单内容时会很有用，可以在发现内容有误时返回 false 以取消表单数据的发送。另外，如果像代码清单 10.1 中的例子这样，在 <a> 标签的 onclick 事件处理程序中返回 false，则会取消页面的跳转。

代码清单 10.1　在事件处理程序中返回 false 值

```
<script>
    function stop(event) {
        alert('Stop page transfer');
        return false;
```

```
    }
</script>
<a id="foo" href="http://example.com" onclick="return stop();">example.com</a>
```

10.2.2 指定为 DOM 元素的属性

如果一个页面分别使用了 HTML 文件和 JavaScript 文件，则应该尽可能少地在 HTML 文件中使用 JavaScript 代码，以提高可维护性。因此，最好将事件处理程序的设定全都写在 JavaScript 内。

事件处理程序可以被指定为节点的属性（代码清单 10.2）。

代码清单 10.2 将事件处理程序指定为属性

```
var btn = document.getElementById('foo');
function sayFoo() {
    alert('foo');
}

btn.onclick = sayFoo;
```

需要注意的是，这里被指定为事件处理程序的正是一个函数。像下面这样，以函数执行后的返回值或用于 HTML 标签的字符串的形式来设定的话，将会发生错误。

代码清单 10.3 事件处理程序的设定

```
btn.onclick = sayFoo();        // 这种方式指定的是函数执行后的返回值，是错误的
btn.onclick = "sayFoo()";      // 以字符串的形式指定该函数也是无效的
bth.coclick = sayFoo;          // 将函数指定为了事件处理程序，而能够正常运行
```

与通过 HTML 标签的属性设定时不同，这里必须全部使用小写字母书写。

而在设定为了属性之后，HTML 标签属性中的内容将会被覆写。因此，如果希望通过 JavaScript 代码在 HTML 标签属性所指定的内容之后再追加新的处理操作，仅采用这种指定 DOM 元素的方法是很难实现的。在 DOM Level 2 Events 中定义的一种方法可以简单地解决这一问题，这种方法将在之后的小节中说明。

10.2.3 通过 EventTarget.addEventListener() 进行指定

■ 注册事件侦听器

虽然之前所介绍的各种方式也能够对事件注册各种各样的处理，但它们都有一个缺点，那就是对于某一个元素的某一个事件，只能够指定 1 种处理操作。

如果只能够指定 1 种处理操作的话，就很难处理复杂的行为。为了弥补这一缺点，在 DOM Level 2 中定义了 EventTarget.addEventListener() 方法（代码清单 10.4）。不过正如前面所说，该方法无法在 Internet Explorer 8 以及更早版本的浏览器中使用。为此可以在 Internet Explorer 中换用 attachEvent() 方法。关于这方面的内容，请参见 8.5 节。

代码清单 10.4 注册事件侦听器

```
var btn = document.getElementById('foo');
btn.addEventListener('click', function (e) {
    alert('foo');
}, false);
```

在注册事件侦听器时，还可以指定第 3 个参数，用以指定从捕获阶段还是从事件冒泡阶段开始执行。这两种阶段将在之后说明。在 DOM Level 2 中，这一参数是必须的。而在 DOM Level 3 中，如果省略了该参数，则会默认从事件冒泡阶段开始执行。之前介绍的指定为 HTML 元素属性的方式，以及指定为 DOM 元素属性的方式，都会在事件冒泡阶段执行事件处理程序。如果希望在捕获阶段执行事件处理程序

的话，则只能使用 EventTarget.addEventListener() 方法了。而在 Internet Explorer 所使用的 attachEvent() 方法中，是没有与之相对应的参数的。在 Internet Explorer 中，事件侦听器总是会在事件冒泡阶段被执行。

■ 事件侦听器的执行顺序

可以通过 addEventListener() 方法对某个特性元素的特定事件设定多个不同的事件侦听器。如果注册了多个事件侦听器，则会产生事件侦听器之间的执行顺序的问题。然而在 DOM Level 2 中并没有对这一点进行定义。在 DOM Level 3 中则是将执行顺序规定为与注册顺序相同。事实上，目前绝大部分的浏览器也都是以注册的顺序对事件侦听器执行的。即使如此，对于和执行顺序有关的处理，还是应该把它们放在同一个事件侦听器中执行，而不应该将它们置于多个不同的事件侦听器之中。

此外，不能同时对事件目标、事件类型及执行阶段都相同的对象注册多个相同的事件侦听器。之后的注册将会被忽略。在这种情况下，事件侦听器的注册顺序不会发生变化，所以其执行顺序也不会改变。

代码清单 10.5　对同一个事件侦听器进行注册

```
var btn = document.getElementById('foo');
function sayHello() {
    alert('Hello');
}
btn.addEventListener('click', sayHello, false);
btn.addEventListener('click', sayHello, false);  // 对相同的事件侦听器进行注册将被忽略
btn.addEventListener('click', sayHello, true);
// 由于执行阶段不同，则将会被作为另一个事件侦听器被注册
```

代码清单 10.6　事件侦听器的执行顺序

```
var btn = document.getElementById('foo');
function sayFoo() {
    alert('foo');
}
function sayBar() {
    alert('bar');
}
function sayBaz() {
    alert('baz');
}
btn.addEventListener('click', sayFoo, false);
btn.addEventListener('click', sayBar, false);
btn.addEventListener('click', sayBaz, false);
btn.addEventListener('click', sayFoo, false);    // 根据规则，这次注册将被忽略

// 在点击按钮时，应该会以 foo、bar、baz 的顺序显示对话框
// 在 Firefox、Google Chrome 以及 Safari 中，将会以预想的情况执行
// 在 Opera 中则将会以 bar、baz、foo 的顺序执行。这是由于第二个 btn.addEventListener('click',
   sayFoo, false) 将会改变事件侦听器的注册顺序
```

■ 事件侦听器对象

通常，只需要使用函数就能够指定事件侦听器。一些浏览器还可以将含有 handleEvent() 方法的对象指定为事件侦听器（代码清单 10.7）。

代码清单 10.7　将对象注册为事件侦听器

```
var btn = document.getElementById('foo');
var eventListener = {
    message: 'This is an event listener object.',
    handleEvent: function (e) {
        alert(this.message);
    }
};
btn.addEventListener('click', eventListener, false);
```

```
// 在点击按键时将会显示一个含有 'This is an event listener object.' 消息的对话框
```

原本在 DOM Level 2 Events 中，EventListener 接口的定义仅仅是一种含有 handleEvent() 方法的一种接口。对于 Java 这类的函数不是第一类对的语言来说，这种定义是有效的。然而，在 DOM Level 2 Events 的附录中对 ECMAScript Language Binding 进行定义时，又规定了 EventListener 对象只是一个函数。

因此，JavaScript 是可以向 addEventListener() 方法传递函数的。DOM Level 3 会对 EventListener 究竟是一个函数还是一个对象进行表述，不过目前还没有定论。所以，可以认为在 JavaScript 中向 addEventListener() 方法传递对象，是一种与 DOM 的定义相背的做法。但是现在主要的浏览器都对这一功能进行了实现，所以如果非要这样使用也不会有什么问题。

还可以将事件对象作为参数传递给一个事件侦听器。之后将对该事件对象说明。

为了便于今后的说明，先在此对两个词进行定义。第一个词是事件目标。这是触发了某个事件的元素，可以通过事件对象的 target 属性对其引用。另一个词是侦听器目标。这是注册了某个事件侦听器的元素，可以通过事件对象的 currentTarget 属性对其引用。

10.2.4 事件处理程序 / 事件侦听器内的 this 引用

在事件处理程序内的 this 所引用的对象即是设定了该事件处理程序的元素。如果是像下面这样的代码，确实是没有什么问题的。

```
document.getElementById('foo').onclick = function () { /* this 是 #foo 元素 */ };
```

然而，下面这种情况则会有些不同。lib 可能会被认为是 this 所引用的对象，但事实上，this 引用的是设定了事件处理程序的元素。

```
var Listener = function () {};
lib.handleClick = function (event) { /* this 引用的是 lib？ */ };
document.getElementById('foo').onclick = lib.handleClick;
// => 在 lib.handleClick 中，this 引用的不是 lib 而是 #foo 元素
```

如果希望在 lib.handleClick 内通过 this 引用 lib，可以像下面这样，先包装一个匿名函数之后再设定。

```
document.getElementById('foo').onclick = function (event) {
    lib.handleClick(event);
    // => 在 lib.handleClick 中的 this 将会引用 lib
};
```

对于事件侦听器来说，上面的情况同样成立。在 JavaScript 中，我们必须对 this 的使用方式十分小心。关于 JavaScript 中 this 引用的替换方法，请参见 6.8.2 节。

10.3 事件的触发

事件的发生主要是由用户操作引起的。在用户浏览网页的过程中，发生最为频繁的事件是 mousemove 事件。在鼠标指针移动的过程中，这一事件将会持续发生。因此，如果设定了 mousemove 事件的处理，则有可能会导致鼠标移动速度变慢。在对 mousemove 事件设定事件处理程序或事件侦听器时，请对这一点多加注意。

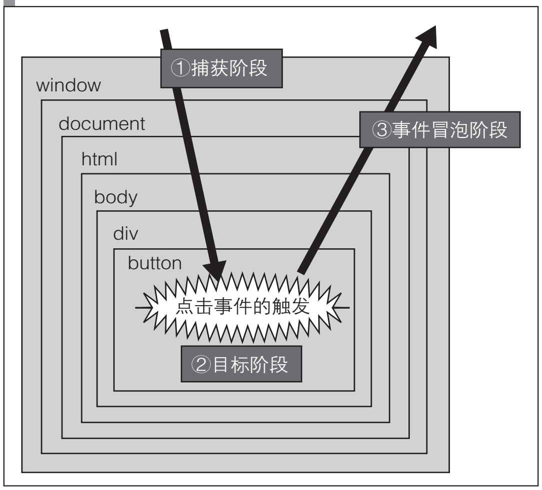
10.4 事件的传播

在浏览器中显示 HTML 文档时，HTML 的元素将会嵌套显示。例如，在下面的例子中，<div> 元素中还有一个 <button> 元素。如果点击了 sample 按钮，则会以该按钮作为事件目标触发一次点击事件。

```
<html>
    <body>
         <div id="foo">
              <button id="bar">sample</button>
         </div>
    </body>
</html>
```

这时，事件的处理将会分别经过捕获阶段、目标阶段、事件冒泡阶段这 3 个阶段（图 10.1）。

图 10.1　事件流程



10.4.1　捕获阶段

在捕获阶段中，事件将会从 Window 对象开始向下遍历 DOM 树来传播。

如果注册了事件侦听器，则会在捕获阶段执行相应的处理。

10.4.2　目标阶段

在这一阶段中，被事件目标注册的事件侦听器将会被执行。如果一个事件处理程序被指定为了 HTML 的标签属性，或被指定为了对象的属性，则会在这一阶段中被执行。

10.4.3　事件冒泡阶段

在这一阶段中，时间的传播方式为从事件目标开始向上遍历 DOM 树，直至 Window 对象结束。在该树的节点中注册的事件侦听器将会在这时被执行。

不过，也有一些事件不会经过冒泡阶段。比如，click 事件为了确定需要涉及哪些元素而有必要在传播事件的过程中遍历 DOM 树，而 focus 事件则是一种只需处理当前元素的事件。这种情况下对 focus 事件进行传播是没有什么意义的，所以 focus 事件不会经过冒泡阶段。

10.4.4　取消

■ 取消传播

还可以取消事件的传播。通过在事件侦听器内执行 Event.stopPropagation() 方法就能取消传播。不过

stopPropagation() 方法只能取消执行在之后的侦听器目标中注册的事件侦听器，而在当前的侦听器目标中设定的其他事件侦听器仍然会被执行。

如果要中止其他的事件侦听器的执行，则需要使用在 DOM Level 3 中引入的 stopImmediatePropagation() 方法。与 stopPropagation() 方法不同，当前侦听器目标中设定的其他事件侦听器的执行也会被中止。由于在 DOM Level 2 中还没有对事件侦听器的执行顺序进行规定，因此也就没有这种能够中止其他事件侦听器的执行的方法。在 DOM Level 3 中已经规定了事件侦听器的执行顺序就是其注册的顺序，所以这一方法就变得有意义了（代码清单 10.8）。

代码清单 10.8　stopPropagation() 与 stopImmediatePropagation() 的区别

```
var btn = document.getElementById('foo');
function sayFoo(event) {
    alert('foo');
    event.stopPropagation();
}
function sayBar(event) {
    alert('bar');
    event.stopImmediatePropagation();
}
function sayBaz(event) {
    alert('baz');
}
btn.addEventListener('click', sayFoo, false);
btn.addEventListener('click', sayBar, false);
btn.addEventListener('click', sayBaz, false);
// 在点击按键之后，将会显示 foo 与 bar 的对话框，baz 的对话框不会被显示
```

■ 取消默认处理

此外，还能够通过 Event.preventDefault() 方法来取消浏览器中默认实现的处理操作。

例如，在默认情况下，点击了 <a> 元素之后，将会跳转至链接页面。而如果在这时执行了 Event.preventDefault() 方法，则不会发生这一行为。preventDefault() 方法的作用等同于让一个被指定为了 HTML 标签属性或 DOM 属性的事件处理程序返回了一个 false 值（代码清单 10.9）。

代码清单 10.9　preventDefault() 方法的使用示例

```
<a id="foo" href="http://example.com">example.com</a>
<script>
    var link = document.getElementById('foo');
    function sayFoo(event) {
        alert('foo');
        event.preventDefault();
    }
    link.addEventListener('click', sayFoo, false);
    // 由于使用了 preventDefault()，默认的行为将被取消，不会发生页面跳转
</script>
```

不过也有一些事件无法通过在事件中使用 preventDefault() 方法来中止。blur 事件就是其中之一，它是一个在焦点移动至其他元素时将被触发的事件。

stopPropagation() 方法与 preventDefault() 方法不仅能够被用于事件冒泡阶段，在其他阶段中也能够使用这些方法。

10.5 事件所具有的元素

可以将在事件侦听器或事件处理程序中发生的事件作为参数再次传给这一事件侦听器或事件处理程序。于是，在事件侦听器中可以根据发生的事件的类型或发生事件的节点来进行分支处理。

一个事件对象是一种实现了 Event 接口的对象。在 Event 接口中定义了多种属性（表 10.2）与方法（表 10.3）。

表 10.2　Event 接口的属性一览

属性	说明
type	事件属性的名称。例如在设定事件侦听器时所用的 click 等名称
target	触发了事件的元素的一个引用。此即在本章中所说的事件目标
currentTarget	注册了现在正在执行处理的事件侦听器的元素。尽管与 target 属性有些相似，但两者是不同的。在捕获阶段与事件冒泡阶段中执行的事件侦听器内，currentTarget 与 target 指向的是不同的节点。此即本章中所说的侦听器目标
eventPhase	用于标识处于事件传播的哪一个阶段
timeStamp	事件的发生时间
bubbles	如果处于事件冒泡阶段则返回 true，否则返回 false
cancelable	如果事件能够执行 preventDefault() 方法则返回 true，否则返回 false

表 10.3　Event 接口的方法一览

方法	说明
stopPropagation()	用于中止事件传播的方法
preventDefault()	用于中止默认行为的方法
stopImmediatePropagation()	用于中止其他事件侦听器的方法。在 DOM Level 3 中被引入

10.6 标准事件

10.6.1　DOM Level 2 中所定义的事件

DOM Level 2 定义的事件类型可以分为以下 4 种类型。表 10.4 ～表 10.7 总结了所定义的事件类型。

- HTMLEvent
- MouseEvent
- UIEvent
- MutationEvent

在表 10.4 ～表 10.7 中，“冒泡”表示事件传播过程中是否会在 DOM 树中经过冒泡阶段，而“默认”则表示是否含有可以通过 preventDefault() 方法取消的默认行为。

表 10.4　HTMLEvent 一览

事件类型	冒泡	默认	触发的时机
load	X	X	文档载入完成后
unload	X	X	文档的载入被撤销（例如页面跳转等情况时）
abort	O	X	图像的读取被中断
error	O	X	发生了错误
select	O	X	在 input 元素或 textarea 元素中选中文本
change	O	X	更改了 input 元素的内容
submit	O	O	提交了表单内容
reset	O	X	重置了表单内容
focus	X	X	元素获得了焦点
blur	X	X	元素失去了焦点
resize	O	X	窗口尺寸发生改变
scroll	O	X	发生了窗口滚动

表 10.5 MouseEvent 一览

事件类型	冒泡	默认	触发的时机
click	O	O	元素被点击
mousedown	O	O	在元素上方按下了鼠标按钮
mouseup	O	O	在元素上方放开了鼠标按钮
mouseout	O	O	鼠标指针从元素上方离开
mouseover	O	O	鼠标指针移动至了元素上方
mousemove	O	X	鼠标指针在元素上方移动

表 10.6 UIEvent 一览

事件类型	冒泡	默认	触发的时机
DOMFocusIn	O	X	元素获取了焦点
DOMFocusOut	O	X	元素失去了焦点
DOMActivate	O	O	元素由于鼠标点击或按键操作后被激活

表 10.7 MutationEvent 一览

事件类型	冒泡	默认	触发的时机
DOMSubtreeModified	O	X	文档内容发生了更改
DOMNodeInserted	O	X	添加了子节点（在添加后被触发）
DOMNodeRemoved	O	X	删除了子节点（在删除前被触发）
DOMNodeInsertedIntoDocument	X	X	向文档中添加了节点（在添加后被触发）
DOMNodeRemovedFromDocument	X	X	从文档中删除了节点（在删除前被触发）
DOMAttrModified	O	X	节点中有属性发生了更改
DOMCharacterDataModified	O	X	节点中的文字数据发生了更改

10.6.2 DOM Level 3 中所定义的事件

DOM Level 3 定义的事件类型可以分为以下几类。

- UIEvent
- FocusEvent
- MouseEvent
- WheelEvent
- TextEvent
- KeyboardEvent
- CompositionEvent
- MutationEvent（不推荐使用）
- MutationNameEvent（不推荐使用）

这里添加了在 DOM Level 2 中所没有的键盘事件。尽管现在还不推荐使用 DOM Level 3 Events，不过各种浏览器中已经提供了对键盘输入事件处理的实现。但必须注意的是，在不同的浏览器中，对键盘事件的处理会有所不同。对于这一点，还将会在之后详述。

表 10.8 ～表 10.14 对除了不被推荐的 MutationEvent 与 MutationNameEvent 以外的事件都进行了总结。其中，“冒泡”表示事件传播过程中是否会在 DOM 树中经过冒泡阶段，“默认”表示是否含有可以通过 preventDefault() 方法取消的默认行为，而“异步”则表示该事件是否是异步执行的。对于“异步”一栏为 X 的事件，所注册事件侦听器将会在完成所有处理后才开始执行下一个处理。如果在循环过程中触发了这样的事件则可能会发生预想之外的情况，请务必对此多加注意。此外，虽然在同一时刻可能会有多个类似的事件被触发（例如 focus 和 focusin 等），不过它们在是否会经过冒泡阶段等方面的行为是不同的。应该根据目的选择合适的事件类型。

表 10.8 Level 3 UIEvent 一览

事件类型	异步	冒泡	默认	触发的时机
DOMActivate	X	O	O	元素被激活（不推荐使用。推荐使用 click 事件）
load	O	X	X	文档载入完成后

（续）

事件类型	异步	冒泡	默认	触发的时机
unload	X	X	X	文档的载入被撤销（例如页面跳转等情况时）
abort	X	X	X	图像的读取被中断
error	O	X	X	发生了错误
select	X	O	X	在 input 元素或 textarea 元素中选定文本
resize	X	X	X	窗口尺寸发生了改变
scroll	O	X	X（*1）	窗口发生了滚动

（*1）仅当在 document 对象中被触发时才会冒泡至 window

表 10.9　Level 3 FocusEvent 一览

事件类型	异步	冒泡	默认	触发的时机
focus	X	X	X	元素获得了焦点
blur	X	X	X	元素失去了焦点
focusin	X	O	X	元素获得了焦点
focusout	X	O	X	元素失去了焦点
DOMFocusIn	X	O	X	元素获得了焦点（不推荐使用。推荐使用 focus 或 focusin 事件）
DOMFocusOut	X	O	X	元素失去了焦点（不推荐使用。推荐使用 blur 或 focusout 事件）

表 10.10　Level 3 MouseEvent 一览

事件类型	异步	冒泡	默认	触发的时机
click	X	O	O	对元素进行了点击
dblclick	X	O	O	对元素进行了双击
mousedown	X	O	O	在元素上方按下了鼠标按钮
mouseup	X	O	O	在元素上方放开了按下的鼠标按钮
mouseenter	X	X	X	鼠标指针移动至了元素上方
mouseleave	X	X	X	鼠标指针离开了元素上方
mouseover	X	O	O	鼠标指针移动至了元素上方
mouseout	X	O	O	鼠标指针离开了元素上方
mousemove	X	O	O	鼠标指针在元素上方移动

表 10.11　Level 3 WheelEvent 一览

事件类型	异步	冒泡	默认	触发的时机
wheel	O	O	O	滚动了鼠标滚轮（*1）

（*1）仅当在 document 对象中被触发时才会冒泡至 window

表 10.12　Level 3 TextEvent 一览

事件类型	异步	冒泡	默认	触发的时机
textinput	X	O	O	输入了字符

表 10.13 Level 3 KeyboardEvent 一览

事件类型	异步	冒泡	默认	触发的时机
keydown	X	O	O	键被按下
keypress	X	O	O	键被按下后又被释放，输入了字符
keyup	X	O	O	放开了被按下的键

表 10.14　Level 3 CompositionEvent 一览

事件类型	异步	冒泡	默认	触发的时机
compositionstart	X	O	O	在 IME 中开始变换
compositionupdate	X	O	X	在 IME 中选择了变换的候选项
compositionend	X	O	X	在 IME 中确定了变换项

DOM Level 3 对事件的触发顺序也做了规定。例如，对于事件类型较多的 MouseEvent 来说，事件的触发顺序如下所示。

■ 鼠标指针处于移动过程中的事件触发顺序

1. mousemove
2. mouseover
3. mouseenter
4. mousemove
5. mouseout
6. mouseleave

■ 双击时的事件触发顺序

1. mousedown
2. mousemove（如有必要）
3. mouseup
4. click
5. mousemove（如有必要）
6. mousedown
7. mousemove（如有必要）
8. mouseup
9. click
10. dblclick

10.7 自定义事件

除了标准中所定义的事件，还能够定义并触发自定义的事件。这时，可以通过 createEvent() 方法来创建一个事件对象，并通过目标节点的 dispatch() 方法来分发这一事件对象。这样一来，就可以对目标节点所指定的事件处理程序或事件侦听器进行调用。在 Internet Explorer 中，我们需要分别以 createEventObject() 方法与 fireEvent() 方法来替代 createEvent() 方法与 dispatchEvent() 方法（代码清单 10.10）。

代码清单 10.10 触发自定义的事件

```
var event = document.createEvent('Events');
event.initEvent('myevent', true, true);
var target = document.getElementById('foo');
target.addEventListener('myevent', function () {
    alert('My event is fired.');
}, false);
target.dispatchEvent(event);
```

initEvent() 方法的第 1 个参数用于指定事件类型。

在使用 dispatch() 方法时需要注意的是，该方法是一个同步执行的方法。它并不会以队列的形式依次逐一执行，而会立即通过相应的事件侦听器开始执行。在执行完之后，dispatchEvent() 方法将会返回相应的事件侦听器的返回值。

不过通过 setTimeout() 方法，我们就能够实现 dispatch() 的异步执行（代码清单 10.11）。

代码清单 10.11 异步执行 dispatchEvent()

```
window.setTimeout(function () {
    target.dispatchEvent(event);
}, 10);
```

由于 dispatchEvent() 是同步执行的，因此也可以通过显式地调用事件侦听器来实现相同的功能。既然如此，为什么还要通过 dispatchEvent() 来触发自定义的事件呢。这是因为通过事件触发的形式可以更容易地添加新的处理操作。相比使用具有了回调函数的自定义函数，通过在 DOM 标准中定义的 addEventListener() 方法来添加新的回调函数的方式通用性更强，而且还能够与其他模块共同使用。

第 11 章　客户端 JavaScript 实践

DOM 操作与事件处理是客户端 JavaScript 的基本内容。只要扎实理解了这些内容，之后的部分就不难掌握了。不过，如果要开发 Web 应用程序，仅靠这些知识是不够的。在此将补充一些与样式的处理、AJAX、表单的处理相关的知识。

11.1　样式

对样式进行操作指的是对页面的外观进行操作，而不是页面的内容。只要设计出合适的样式，就能够创建出易读、清晰、美观的 Web 应用程序。话虽如此，易读和美观主要由设计师负责，属于静态设计的范畴，需要 CSS 方面的知识。而借助 JavaScript，则能够动态地变更样式，增强页面的可读性。

JavaScript 实现了动态样式变更，其目的是为用户提供视觉反馈。例如，针对可被点击的元素，如果在鼠标指针移动至该元素上方时改变其图标，并改变 DOM 元素的颜色，就能够向用户传达该元素可被点击的信息。当鼠标指针位于 DOM 元素之上时，将触发 mouseover 事件。只需准备一个能够改变鼠标指针以及 DOM 元素色彩的事件侦听器，并将其注册至该事件，就能够实现这一效果。

11.1.1　样式的变更方法

可以通过以下这些方法对样式进行变更。

- 通过 className 属性更改 class 名
- 通过 classList 属性更改 class 名
- 更改 style 属性
- 直接更改样式表

以上任意一种方法都能够实现样式的变更。根据不同的用途选择合适的方法即可。

■ 通过 className 属性更改 class 名

通过更改 DOM 元素的 class 名来改变样式是一种最为简单的做法。即事先通过 CSS 定义好对应于更改前与更改好的 class 名的样式，并在 JavaScript 中替换 class 名。可以通过 className 属性设定 class 名，如代码清单 11.1 所示。在这个例子中，当点击时，字符的颜色与背景色将会调换。

代码清单 11.1　通过 className 属性更改 class 名

```
<!DOCTYPE HTML>
<html lang="zh-CN">[1]
    <head>
        <meta charset="UTF-8">
        <title>更改 class 名</title>
        <style>
            .foo-before {
```

[1] 此处进行了本地化处理将语言属性改为了中文。下同。——译者注

```
                background-color: white;
                color: black;
            }
            .foo-after {
                background-color: black;
                color: white;
            }
        </style>
    </head>
    <body>
        <div id="foo" class="foo-before">Click me.</div>
        <script>
            var foo = document.getElementById('foo');
            foo.onclick = function toggleStyle() {
                this.className = (this.className === 'foo-before') ? 'foo-after' : 'foo-before';
            };
        </script>
    </body>
</html>
```

在更改 class 名时应该注意并清楚了解的是，如果更改了 class 名，到底有哪些元素将会受到影响。例如对于代码清单 11.2 的情况，如果更改了 1 个元素的 class 名，其相邻元素与子孙元素的样式也会一起改变。这时，如果需要改变的元素过多，则可能出现性能上的问题。不过，如果要考虑性能的话，指定了相邻元素与子元素的 CSS 的写法本身就是有问题的。最好对这些元素分别设定不同的 class 名并进行式样操作。

代码清单 11.2　在更改 class 名后相关元素的样式也会改变

```
<!DOCTYPE HTML>
<html lang="zh-CN">
    <head>
        <meta charset="UTF-8">
        <title>在变更 class 名后相关元素的样式也会改变</title>
        <style>
            .foo-before {
                background-color: white;
                color: black;
            }
            .foo-before p {
                text-decoration: none;
            }
            .foo-before + div {
                text-decoration: none;
            }

            .foo-after {
                background-color: black;
                color: white;
            }
            .foo-after p {
                text-decoration: underline;
            }
            .foo-after + div {
                text-decoration: line-through;
            }
        </style>
    </head>
    <body>
        <div id="foo" class="foo-before">
            <p>one</p>
            <p>two</p>
            <p>three</p>
            <p>four</p>
        </div>
        <div>
```

```
                This is sample text.
            </div>
            <script>
                var foo = document.getElementById(,foo');
                foo.onclick = function toggleStyle() {
                    this.className = (this.className === ,foo-before') ? ,foo-after : ,foo-before';
                };
            </script>
        </body>
    </html>
```

■ 通过 classList 属性更改 class 名

还可以通过对在 HTML5 中新增的 classList 属性进行操作，来更改 class 名。与通过对 className 属性进行操作以更改 class 名相比，这种方法更加容易理解（代码清单 11.3）。

代码清单 11.3　在更改 class 名后相关元素的样式也会改变

```
<script>
    var foo = document.getElementById('foo');
    foo.onclick = function toggleStyle() {
        this.classList.toggle('foo-after');
        this.classList.toggle('foo-before');
    };
</script>
```

表 11.1 总结了可以使用 classList 属性的方法。classList 属性是一种对 DOM TokenList 接口的实现。

表 11.1　可以使用 classList 属性的方法

方法名	说明
contains(clazz)	判断在 class 名中是否含有 clazz
add(clazz)	向 class 名中添加 clazz
remove(clazz)	从 class 名中删除 clazz
toggle(clazz)	如果在 class 名中含有 clazz 则将它删除，否则向 class 名中添加 clazz

■ 更改 style 属性

可以直接更改 DOM 元素的 style 属性的值以实现样式的更改。这与更改 class 名的情况不同，样式更改的范围被明确地限定于这个元素。另外，通过 style 属性指定的内容的应用优先级将仅次于 CSS 中被标记为 !important 的元素。

style 属性中的各个属性名是由在 CSS 中所指定的属性名演变而来的，其更改之处为去除了属性名中的连字符（-）并将连字符之后的字母改为了大写形式。例如，对于 CSS 中的 margin-top 属性，在 JavaScript 中则可以通过 marginTop 的名称使用。之所以要将连字符去除，是因为连字符在 JavaScript 将被识别为减号。此外，由于 float 在 JavaScript 中是作为保留字使用的，因此 float 属性不能直接使用。如果要在 JavaScript 中更改 float 属性，则需要通过 cssFloat 属性来进行操作。

在代码清单 11.4 中是一个更改 style 属性更改的例子。

代码清单 11.4　更改 style 属性

```
<!DOCTYPE HTML>
<html lang="zh-CN">
    <head>
        <meta charset="UTF-8">
        <title>更改 style 属性</title>
        <style>
        #foo {
            background-color: white;
            color: black;
```

```
        }
        </style>
    </head>
    <body>
        <div id="foo">This is foo. Click me.</div>
        <div id="bar">This is bar.</div>
        <script>
            var foo = document.getElementById('foo');
            foo.onclick = function toggleStyle() {
                var style = this.style;
                if (!style.cssFloat) {
                    style.cssFloat = 'left';
                    style.backgroundColor = 'black';
                    style.color = 'white';
                } else {
                    style.cssFloat = '';
                    style.backgroundColor = 'white';
                    style.color = 'black';
                }
            };
        </script>
    </body>
</html>
```

在更改 style 属性时，应当要注意的是，功能和设计是无法分离的。试考虑仅更改色彩的情况。这时，如何对 style 属性的更改进行处理是一个功能方面的问题，而应该如何更改色彩则是设计方面的问题，并不是功能方面的问题。从功能上来说，只需要改变其样式就能够实现目的了。所以说，在 JavaScript 中我们只需要考虑如何实现对样式的更改即可，至于具体应该如何更改样式，则应该由 CSS 负责。

事实上，如果从保守的角度或团队开发的角度来看待网页开发，将设计部分与功能部分分离会带来多种好处。结构由 HTML 实现，功能由 JavaScript 实现，设计则由 CSS 实现。将三部分代码分开书写能够使整体条理变得清晰易懂。

■ 直接更改样式表

还可以直接对是否应用样式表进行设定。如果将 link 元素与 style 元素的 disabled 属性设为 true，相应的样式表就将被禁用（代码清单 11.5）。

代码清单 11.5　直接更改样式表

```
<!DOCTYPE HTML>
<html lang="zh-CN">
    <head>
        <meta charset="UTF-8">
        <title>直接更改样式表</title>
        <link rel="stylesheet" type="text/css" href="style-a.css" id="style-a" disabled="true">
        <link rel="stylesheet" type="text/css" hred="style-b.css" id="style-b" disabled="true">
        <style id="style-c" disabled="true">
            #foo {
                background-color: #999;
            }
        </style>
        <script>
            function change(id, enable) {
                // 在勾选了复选框之后启用样式
                document.getElementById(id).disabled = !enable;
            }
            window.addEventListener('load', function () {
                // 在初始化处理中禁用所有的样式
                var styles = document.styleSheets;
                for (var i = 0; i < styles.length; i++) {
                    styles[i].disabled = true;
                }
            }, false);
```

```
        </script>
    </head>
    <body>
        <div id="foo">This is a sample.</div>

        <input type="checkbox" onchange="change('style-a', this.checked)">
        <input type="checkbox" onchange="change('style-b', this.checked)">
        <input type="checkbox" onchange="change('style-c', this.checked)">
    </body>
</html>

<!--
/* style-a.css 的内容 */
#foo {
    font-size: x-large;
}

/* sytle-b.css 的内容 */
#foo {
    text-decoration: underline;
}
-->
```

需要更改整个页面的样式时，就可以采用这种切换样式表启用禁用状态的方式。例如，对于已经事先准备了一些样式主题，需要让用户选择自己喜欢的主题以显示页面内容的情况，就可以通过这种方式实现。

如果不需要切换整个主题，则没有必要专门通过启用或禁用样式表来实现样式的更改。直接更改 class 名，或更改 style 属性的值即可。

11.1.2　位置的设定

在更改样式时，诸如文字大小或背景颜色等方面的更改操作理解起来并不难，但如果要更改 DOM 元素的位置，则必须知道更多的信息。学会如何将 DOM 元素设置于任意的位置，将有助于实现复杂的网页布局。

在设定位置时，最为关键的两点是 position 属性与鼠标指针的位置。另外则是要对如何处理 DOM 元素的宽度和高度有所了解。

■ position 属性

position 属性可以被指定为以下几种值之一。

- static
- fixed
- absolute
- relative

如果在此设定了 static 以外的值，就能够设置 top、bottom、left、right 属性。

■ static

position 属性的默认值。将根据 HTML 中所写的标签来决定元素的配置。这种情况下无法通过 top 或 left 之类的属性来指定元素的位置。

■ fixed

如果将 position 属性指定为 fixed，则将会以浏览器窗口为基准来确定元素的相对位置。由于是相对于浏览器窗口的位置，因此即使对页面进行了滚动操作，元素在画面上的位置也不会发生改变。也就是说元素将始终保持在同一位置。

该值无法在 Internet Explorer 6 中使用。

■ absolute

如果将 position 属性指定为了 absolute，则可以对该元素与含有该元素的元素之间的相对位置进行设定。通常情况下都是元素与 body 元素之间的相对位置，不过如果嵌套了非 static 值的其他元素，则将以

该元素为基准确定相对位置。

■ relative

如果将 position 属性指定为了 relative，则会在根据 HTML 中所写的标签进行配置的基础上，对元素的相对位置进行设置。

不过，如果已经指定了 relative 值，一般也就不会再设置 top 或 left 之类的值了。通常的做法是，以被设定为 absolute 的元素为标准来进行位置设定。根据对 absolute 的说明可以知道，被设定为 absolute 属性的元素，是以含有该元素的元素中 position 属性没有被设定为 static 的元素为基准来确定位置的。而被设定为 relative 的元素的位置与其被设定为 static 时的情况并没有什么不同，将会被动态地置于正确的位置。这样一来，就能够按照期望的那样，根据其与被设定为 relative 值的元素的相对位置来配置该元素的位置。

11.1.3 位置

如果要实现在鼠标点击位置附近显示一个框这样的效果，则必须要知道点击位置。在 MouseEvent 被触发时，相应的 Event 对象提供了多个属性来获取这一位置。这时的问题在于，需要知道点击的位置是以什么为基准来表示的。

```
function onclick(event) {
    // 通过 event 对象获取鼠标指针的位置，并进行相应的处理
}
```

■ 屏幕坐标

可以通过 screenX 和 screenY 属性来获取屏幕坐标。屏幕坐标是一种以计算机显示器的左上角为原点的坐标系。不过由于这只是屏幕上的位置坐标，因此能够对其进行有效利用的情况很少。

■ 窗口坐标

可以通过 clientX 和 clientY 属性来获取窗口坐标。窗口坐标是一种以浏览器的显示范围的左上角为原点的坐标系。这一坐标系与文档、元素的滚动情况无关，其坐标值仅由内容的显示位置决定。

■ 文档坐标

可以通过 pageX 和 pageY 属性来获取元素在文档中的位置。文档坐标是一种以文档页面的左上角为原点的坐标系。与窗口坐标不同，其坐标值与显示位置无关，是由元素在整个文档中的位置决定的。

DOM 对 screenX 与 clientX 进行了定义，但没有定义 pageX。这一坐标系是由浏览器自定义实现的。Internet Explorer 8 以及更早版本无法使用这一坐标。

■ 在特定元素内的相对坐标

通过 layerX 与 layerY，或 offsetX 与 offsetY 属性，可以获取触发了事件的元素内的相对坐标。这些属性没有在 DOM 中被定义，是由浏览器自定义实现的功能。

Element.getBoundingClientRect() 虽然不能直接取得相对坐标，但也能实现类似的功能。该方法是在 CSSOMView Module 这一与文档显示方式有关的标准中被定义的。通过使用 getBoundingClientRect()，能够获取元素范围信息的窗口坐标。这里的范围信息指的是，距离左侧的距离（left）、距离上方的距离（top）、宽度（width）与高度（height）。可以像代码清单 11.6 中这样，将这些值与 clientX 和 clientY 结合使用以获取元素内的相对坐标。

而代码清单 11.7 是一个完整的例子，它将在点击的位置显示元素。

代码清单 11.6 在元素内取得相对坐标

```
function onclick(event) {
    var x = event.clientX;     // 窗口坐标系中鼠标指针的 x 坐标
```

```
    var y = event.clientY;     // 窗口坐标系中鼠标指针的 y 坐标
    var r = event.target.getBoundingClientRect();    // 窗口坐标系中被点击元素的范围信息
    x -= r.left;               // 鼠标指针在被点击元素内部的 x 坐标
    y -= r.top;                // 鼠标指针在被点击元素内部的 y 坐标
}
```

代码清单 11.7　一个在点击位置显示元素的例子

```
<div id="foo" style="width: 2000px; height: 2000px; position: relative;">
    <div id="message" style="position: absolute; background: lightgray; width: 100px;">Hello, world!</div>
</div>
<script>
var foo = document.getElementById('foo');
function getPosition(event) {
    var x = event.clientX;
    var y = event.clientY;
    var r = event.target.getBoundingClientRect();
    x -= r.left;
    y -= r.top;
    return { x: x, y: y };
}
foo.addEventListener('click', function (event) {
    var message = document.getElementById('message')
    if (event.target === message) {
        // 如果点击 message，则不进行操作
        return;
    }
    var pos = getPosition(event);
    message.style.left = pos.x;
    message.style.top = pos.y;
}, false);
</script>
```

11.1.4　动画

可以通过动画使样式以一定的速率逐渐变化。如果要实现移动的动画效果，则可以对被设定为 position:absolute 的元素的 left 属性进行渐变。如果要实现淡入淡出的效果，则可以逐渐改变元素的透明度（opacity）的值。

为了使样式可以以一定的速率逐渐变化，自然就需要让 JavaScript 定期执行。setInterval 是一个能够定期执行 JavaScript 的函数。在对 setInterval 指定了某个值之后，每经过所指定的时间，就会执行一次特定的函数。代码清单 11.8 是一个使用示例。

代码清单 11.8　动画

```
<div id="foo" style="position: absolute">This is a sample.</div>
<script>
var elem = document.getElementById('foo');
var frame = 0;
setInterval(function () {
    frame += 1;
    elem.style.left = frame * 10 + 'px';
}, 100);          // 每经过 100 毫秒将向右移动 10 个像素
</script>
```

此外，就算不使用 JavaScript，仅通过 CSS3 也能够实现各种各样的动画效果。虽然实际的性能和浏览器的具体实现有关，不过一般来说，与通过 JavaScript 实现的动画相比，通过 CSS 实现的动画性能更好。如今很多智能手机中都采用了支持 CSS3 的浏览器，由于上述性能原因，如果网页运行于这些浏览器中的话，则应该更多地利用 CSS 来实现动画效果。

11.2 AJAX

AJAX 是 Asynchronous JavaScript + XML 的简称[①]。AJAX 一词的实际含义为“不发生页面跳转、异步载入内容并改写页面内容的技术”。在实际操作中，AJAX 不仅仅会使用 XML 数据，很多时候也会对 JSON 或纯文本进行操作。AJAX 这个词是由 Jesse James Garrett 在 2005 年命名的。不过，在那之前就已经有使用 AJAX 的网站。众所周知，Google 的 Gmail 就是一个利用了 AJAX 的优秀范例。如今，AJAX 正在全世界范围内迅速地得到普及。

11.2.1　异步处理的优点

AJAX 的关键在于它是以异步的方式执行的。异步处理的优点是不会让用户白白等待。对于同步处理来说，在处理完来自服务器的响应之前，用户无法进行任何其他操作，只能等待。如果服务器的响应发生了延迟，会让用户误以为页面失去了响应。在优先考虑用户体验时，与同步处理相比，采用异步处理的方式更为合适，这一点是显而易见的。JavaScript 是一种事件驱动程序设计语言，在很多地方都会用到异步处理，所以要理解异步处理并不是一件难事。

11.2.2　XMLHttpRequest

如果要通过 JavaScript 动态地向服务器发送请求，则需要使用 XMLHttpRequest 对象。不过在这里要稍微提醒一下大家，XMLHttpRequest 目前尚未被制定为标准。之所以说是“稍微”，是因为那些现代的浏览器自然不必多说，即使是 Internet Explorer 也在 7 以及之后的版本里全都使用了通用的 API，因此它已经成为了一种事实标准。考虑到这点，尚未标准化也就不是什么问题了。问题在于 Internet Explorer 6。不过它也仅仅是在对象的创建方式上有所不同，对象所含有的方法与其他浏览器中的实现是通用的。可以像代码清单 11.9 这样，在 Internet Explorer 6 中创建 XMLHttpRequest 的替代对象。

代码清单 11.9　XMLHttpRequest 的跨浏览器支持

```
if (!window.XMLHttpRequest) {
    // Internet Explorer 6
    XMLHttpRequest = function () {
        var objs = ['MSXML2.XMLHTTP.6.0', 'MSXML2.XMLHTTP.3.0', 'MSXML2.XMLHTTP', 'Microsoft.XMLHTTP'];
        for (var i = 0; i < objs.length; i++) {
            var obj = objs[i];
            try {
                return new ActiveXObject(obj);
            } catch (ignore) {
            }
        }
        throw new Error('Cannot create XMLHttpRequest object.');
    }
}
var xhr = new XMLHttpRequest();
```

11.2.3　基本的处理流程

接下来我们来说明使用 XMLHttpRequest 时的基本处理流程。

① 亦可采用仅首字母大写的 Ajax 写法。——译者注

■ XMLHttpRequest 对象的创建

在上一节中，我们已经说明了 XMLHttpRequest 对象的创建方法。在创建了 XMLHttpRequest 对象之后，就需要对服务器的 URL 进行指定，以发送请求（代码清单 11.10）。

代码清单 11.10　发送请求

```
var xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {
    if (xhr.readyState == 4) {
        if (xhr.status == 200) {
            alert(xhr.responseText);
        }
    }
};
xhr.open('GET', 'http://example.com/something');
xhr.setRequestHeader('If-Modified-Since', 'Thu 01 Jun 1970 00:00:00 GMT');
xhr.send(null);
```

onreadystatechange 这一事件处理程序将会在 XMLHttpRequest 对象的状态发生变化时被调用。用于表示状态的 readyState 的取值范围为 0~4，其中 4 表示已经完成了对来自服务器的响应的接收处理。表 11.2 解释了 0~4 所表示的含义。

表 11.2　readyState 的含义

readyState	含义
0	open() 尚未被调用
1	send() 尚未被调用
2	服务器尚未返回响应
3	正在接收来自服务器的响应
4	完成了对来自服务器的响应的接收

在 status 中包含了响应的状态码。如果通信正常结束，该值为 200。关于 HTTP 响应的状态码的详细信息，请参考其他相关书籍。

在 responseText 中包含了服务器响应的字符串形式。而对于 XML 的情况，响应会以 DOM 对象的形式包含于 responseXML 之中。根据情况选择合适的响应类型即可。不过无论响应的形式如何，在 responseText 中都会含有相应的值，所以只要使用 responseText 的值就不会有什么问题了。本来，如果一个响应是 XML 格式的，自然是应该引用 responseXML 会更好，不过现在流行的做法是通过 JSON 通信，所以需要使用 XML 的情况也变少了。通过 JSON 进行通信的方式所需要的传送数据量较少，对数据的处理也更方便，因而也很常见。这部分处理和服务器的情况也有关，可能会有各种各样的限制。但如果可能，还是推荐采用 JSON 格式来传输数据。

传递给 open() 的参数是 HTTP 请求类型及通信目标服务器的 URL。

在这个例子中，仅传递了两个参数给 open()。但实际上它还能再接受 3 个参数。

如果传递了 false 值至第 3 个参数，XMLHttpRequest 就会执行同步通信。该参数的默认值为 true，也就是将执行异步通信。如果是同步通信的话，自然在此期间无法执行其他操作。对于用户的可操作性来说，执行同步通信几乎不具有任何优点。第 4 个参数与第 5 个参数分别是用户 ID 和密码。在向需要进行身份认证的服务器发送请求时要用到这两个参数。

从 open() 这一方法名就可以知道，其作用只是创建与服务器的连接，并不具备其他的功能。它只是保存了 HTTP 请求类型以及服务器 URL 而已。

之后的 setRequestHeader() 用于请求头部的设置。在通信的目标服务器中会自动发送对应的 Cookie，所以并不需要显式地设置。当然也可以显式地设定一个不同的值。

实际向服务器发送请求的是 send()。如果是 POST 请求类型，则会将参数所收到的数据发送至服务器。如果是 GET 请求类型或 HEAD 请求类型等不需要发送数据的 HTTP 请求类型，则参数为 null。

11.2.4 同步通信

接下来，我们说明一下如何通过 XMLHttpRequest 进行同步通信。如果要执行同步通信，则不必对 onreadystatechange 事件处理程序进行设定。在执行了 send() 之后该处理将会进入待机状态，只要在 send() 之后继续书写对响应的处理操作即可（代码清单 11.11）。

代码清单 11.11 通过 XMLHttpRequest 进行同步通信

```
var xhr = new XMLHttpRequest();
xhr.open('GET', 'http://example.com/something', false);
// 将第 3 个参数指定为 false 的话就会执行同步通信
xhr.setRequestHeader('If-Modified-Since', 'Thu, 01 Jan 1970 00:00:00 GMT');
xhr.send(null);       // 此时，客户端侧的处理将会进入待机状态
// 执行至此时已经完成了对响应的接收
if (xhr.status === 200) {
    alert(xhr.responseText);
}
```

在进行同步通信时，代码顺序和实际的通信过程是对应的，理解起来会更为容易。不过在实际操作中通常不会使用这种方式。在客户端 JavaScript 中最重要的是如何减少用户的等待时间，尽可能向用户提供流畅的使用体验。所以，应当避免使用同步通信来减少等待时间。

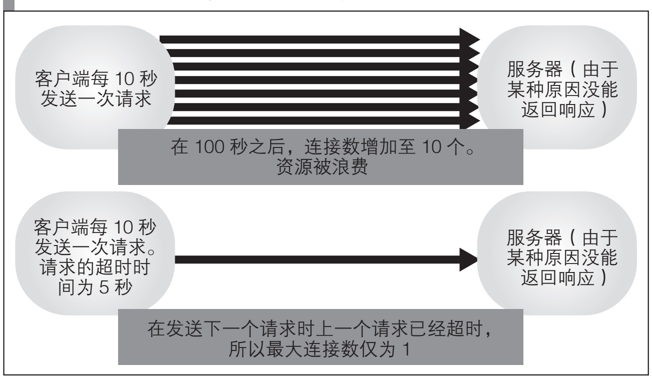
11.2.5 超时

在进行同步通信时，如果通信过程很费时，处理操作则会在 send() 处等待，其他的处理将会无法进行。这时，应当在长时间无法取得响应的情况下取消该请求。这种情况称为请求超时。

XMLHttpRequest 基本上采用的都是异步通信，所以无论通信需要花费多少时间，都不会影响用户操作。不过有些时候，最好设置合适的超时时间。

例如，有时会需要一定的时间间隔发送请求以改写页面的内容。如果在响应返回之前就继续发送下一个请求，就会产生大量通信。这对于客户端与服务器来说都不是一件好事。对于这个问题，专门设计一种实现，使其只有在收到前一个请求的响应之后才发送下一个请求，也是一种不错的做法。不过其实只要简单地对超时时间进行设置，就能够避免同时产生多个请求的问题了（图 11.1）。

图 11.1 通过超时来限制连接数的增加



如果要取消请求，其实只需要执行 abort() 方法即可。而通过 setTimeout() 方法可以在一定时间后执行 abort() 方法，从而实现超时功能。另外还有一个 clearTimeout() 方法，如果在一定时间内收到了返回的响应，该方法就将会取消 abort() 的执行（代码清单 11.12）。

代码清单 11.12　超时处理

```
var xhr = new XMLHttpRequest();
var timerId = window.setTimeout(function() {
    xhr.abort();
}, 5000);        // 5 秒后将会超时

xhr.onreadystatechange = function() {
    if (request.readyState === 4) {
        // 取消超时处理
        window.clearTimeout(timeId);
    }
};
```

11.2.6　响应

■ 通用类型的响应

可以通过 responseText 属性来引用一个 XMLHttpRequest 响应。即使目标的 Content type 不是 text/plain，也能够以该属性进行设定。这时所使用的是响应的 body 部分的内容。例如，如果收到的是一个 HTML，则可以选择将某个元素的 innerHTML 属性设定为 responseText 的内容（代码清单 11.13）。

代码清单 11.13　通用类型的响应

```
var xhr = XMLHttpRequest();
// ...
var dom = document.getElementById('foo');
foo.innerHTML = xhr.responseText;
```

■ XML 类型的响应

从 XMLHttpRequest 这一名称也能看出，它能够以 XML 的形式接收 XMLHttpRequest 的响应。在接收时，最好使用 responseXML 属性而不是 responseText 属性。因为 responseXML 属性能够对 XML 的解析结果进行引用（代码清单 11.14）。

代码清单 11.14　XML 类型的响应

```
var xhr = XMLHttpRequest();
// ...
var xml = xhr.responseXML;

// 假定 xml 的内部是这样的内容
// <result>
//   <apiversion>1.0</apiversion>
//   <value>foo</value>
//</result>
alert(xml.getElementsByTagName('value')[0].firstChild.nodeValue);  // => foo
```

■ JSON 形式的响应

最近，返回 JSON 的 API 越来越多了。XML 的书写过于冗长，响应的体积也越来越大。而且在 JavaScript 中对 XML 进行操作也需要书写冗长的代码。于是，相对于 XML 而言，更易于处理、书写也更简单的 JSON 受到了欢迎。

如果有一个与 responseXML 属性类似的 responseJSON 属性的话，一切就很方便了，可惜的是，并不存在这样的属性。如果要接收 JSON，就必须将 responseText 属性的内容转换为 JSON。这时，需要使用的是 JSON.parse() 方法（代码清单 11.15）。

代码清单 11.15 JSON 类型的响应

```
var xhr = XMLHttpRequest();
// ...
var json = JSON.parse(xhr.responseText);

// 假定json的内部是这样的内容
// {
//     "apiversion": 1.0,
//     "value": "foo"
// }
alert(json.value);    // => foo
```

在浏览器对 JSON.parse 实现之前，是通过 eval 来对 responseText 求值的。如果 JSON 字符串的内容不正确，有可能导致页面数据的损坏或被错误地改写。为了避免发生这种情况，应该选择使用 JSON.parse() 方法。在 Internet Explorer 7 以及更早的版本中，没有对 JSON.parse() 方法进行实现。所以在这时应该通过 http://www.json.org/ 的 json2.js 等方式来实现安全的 JSON 分析。

11.2.7 跨源限制

所谓跨源限制指的是，对源不同的通信进行限制。而这里的源指的是由 URL 的协议（http: 或 https: 等）、主机名、端口号所构成的元素。在 Web 领域，为了确保安全性，只有同源的通信才能被允许进行，这称为同源策略。

虽然可以在 HTML 中使用 iframe 以实现在一个页面中同时显示来自不同域的文档，不过 JavaScript 仍然只能访问同一个源的文档。如果文档的 URL 和 iframe 的不同，则无法通过文档中所包含的 JavaScript 对 iframe 内的 DOM 进行操作，而 iframe 内的 JavaScript 也无法操作文档中的 DOM。如果不这样，就会发生诸如不同域的 Cookie 能够相互访问等安全问题。

对于 XMLHttpRequest 来说，同源策略的含义是，一个 XMLHttpRequest 对象只能发送至一个特定的服务器，即提供了使用该 XMLHttpRequest 对象的文档的下载的那个服务器。不过，只要让服务器转发该请求，就能够将请求发送至不同域的服务器。

11.2.8 跨源通信

跨源通信指的是在不同的源之间收发请求。但是，由于前面所说的同源策略的存在，一般情况下是无法通过 XMLHttpRequest 向不同的源发送请求的。既可以通过服务器转发或 Flash 来实现跨源请求的发送，也可以通过 JavaScript 实现跨源通信。

有时还会使用跨域通信这一术语，它的意义和跨源通信类似。前面提到过，所谓源，是一种由协议、域、端口号组成的元素。因此，可以说跨域通信是跨源通信的一个子集。

以下这些方法都能够实现在 JavaScript 中的跨源通信。

- JSONP
- iframe 攻击（iframe hack）
- window.postMessage()
- XMLHttpRequest level 2

接下来我们将对它们进行说明。

11.2.9 JSONP

虽然无法通过 XMLHttpRequest 进行跨源通信，但是可以将 script 标签的 src 属性指定为其他域中

的 JavaScript 文件并将其载入。如果在此处动态地创建 script 标签的话，就能实现对其他域中的数据的动态读取。不过，如果仅仅是取得了数据，还无法在客户端使用。因此产生了 JSONP 这一概念。这是由 MochiKit 这一 JavaScript 库的作者 Bob Ippolito 提出的[①]。

JSONP 是 JSON with Padding 的简称。这里的 Padding 指的是向 JSON 数据中添加函数名。

此时，服务器会像下面这样对数据添加函数名之后将其返回。

```
callback({
    "foo": "This is foo",
    "bar": "This is bar",
    "baz": "This is baz"
});
```

当客户端对其求值时，如果在客户端侧对该函数进行了定义（在上面的例子中，指的是 callback 的函数），就能够执行这个函数。也就是说，JSONP 的思想是不但能从服务器处获取数据，而且还能将这些数据作为函数的参数使用。这种方式能够实现 JavaScript 的跨源通信，因而被用于各种网页之中。

代码清单 11.16 是一个 JSONP 的使用示例。

代码清单 11.16　JSONP 的使用示例

```
<script>
function foo(json) {
    // 使用 json 数据进行一些操作
}

function loadData() {
    var elem = document.createElement('script');
    // 将 foo 指定为所要执行的回调函数
    // 在使用 JSONP 的 API 中，常常可以对 callback 函数的名称进行指定
    elem.src = 'http://api.example.com/some-data&callback=foo';
    // 将 script 标签添加至 head 中
    // 这时 DOM 将被重建，并载入 script 标签的 src 的内容
    // 载入之后就会执行 foo 函数
    document.getElementByTagName('head')[0].append(elem);
}
</script>
```

JSONP 存在的一个问题是它无法在 POST 请求类型中使用。这时只能够做到动态创建 script 标签并读取数据，而无法从客户端发出数据。如果想让客户端以 POST 的形式发送跨源数据，则必须通过其他手段。

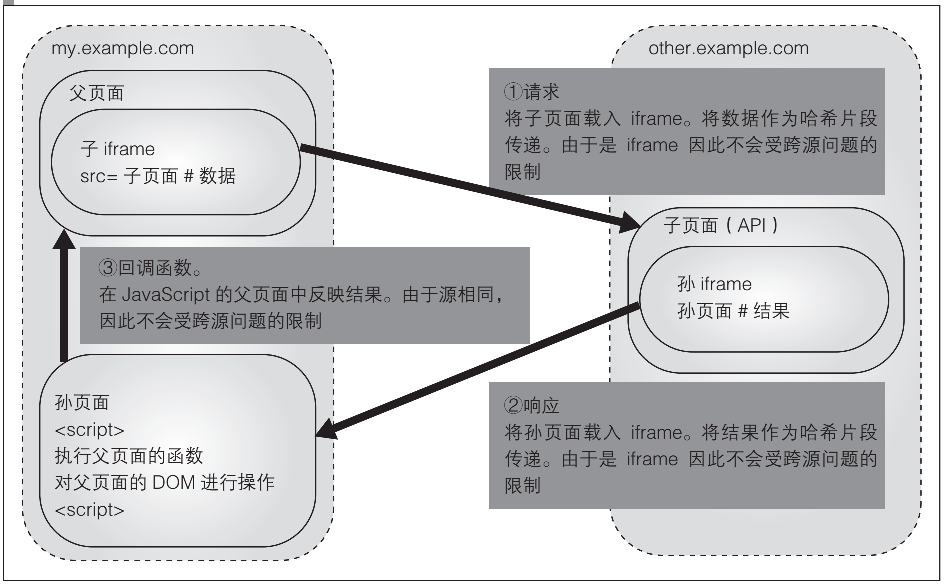
11.2.10　iframe 攻击（iframe hack）

通过 iframe 来进行跨源通信的原理稍微有些复杂，需要用到下面一些概念。括号内表示的是各自所属的域。其中，父页面与孙 iframe 必须是相同的域。由于存在同源策略，因此无法对不同域的 iframe 内的 DOM 元素进行操作，也无法在一个 iframe 中对其父节点的 DOM 元素进行操作。不过，如果父页面与 iframe 的源相同的话，则可以相互对对方的 DOM 进行操作（图 11.2）。

- 父页面（my.example.com）
- 子 iframe（other.example.com）
- 孙 iframe（my.example.com）

① http://bob.pythonmac.org/archives/2005/12/05/remote-json-jsonp/

图 11.2 通过 iframe 攻击实现跨源通信



■ API 请求

首先，在 my.example.com 的页面中将会指定一个 other.example.com 中的 html 来创建一个 iframe。这里的关键点在于 URL 中包含了哈希片段。哈希片段被指定为了进行 API 调用时所需的数据。

■ 响应

在 other.example.com 的页面中，将会通过使用 XMLHttpRequest 之类的方式调用 other.example.com 中的功能并获取数据。这时还没有进行跨源的功能调用，必须在取得数据后再将它们传回之前的 my.example.com 中的页面。为此，需要在 other.example.com 的页面内创建一个 iframe，并将其指向 my.example.com 中的页面。即孙 iframe。这一孙 iframe 的 URL 也要被指定为哈希片段。和在父页面中创建的指向 other.example.com 的子 iframe 一样，数据将被置于哈希片段之中。

■ 回调函数

由于孙 iframe 与父页面都是 my.example.com 中的页面，因此可以在孙 iframe 中执行父页面中的函数。这时，孙 iframe 的 onload 将会调用父页面的函数，从而实现 callback 的调用。当然了，在调用函数时所使用的参数是从子 iframe 传来的 location.hash 的值。至于子 iframe 将会调用哪一个页面，则是在创建子 iframe 时通过哈希片段来指定的。只要在创建时将所指定的这一页面置于子 iframe 内即可。如此一来，从父页面的角度来看，就实现了与跨源通信相同的执行结果。

代码清单 11.17 ～ 11.19 是以上内容的范例代码。

代码清单 11.17 通过 iframe 实现跨源通信（父页面）

```
<html>
    <head>
        <title> 父页面 </title>
        <script>
        // 在跨源通信中用于获取数据的函数
        function getData() {
            // 此处子 iframe 的 URL 为 other.example.com 的页面
            // 参数则是在 # 之后的数据
            frames[0].location.href =
```

```
                    'http://other.example.com/api.html#' +
                    '{' +
                        // 这里是事实上希望执行的 API
                        '"api": "http://other.example.com/some-data",' +
                        // 在子 iframe 中指定的孙 iframe 的 URL
                        '"callback": "http://my.example.com/callback.html"' +
                    '}';
            }

            // 在跨源通信中作为回调函数被执行的函数
            // 由孙 iframe 调用
            function callback(param) {
                document.getElementById("result").innerHTML = param;
                frames[0].frames[0].location.href = 'dummy.gif';
            }
            </script>
        </head>
        <body>
            <input type="button" value=“从 other.example.com 获取数据” onclick="getData()">
            <div id="result"></div>
            <iframe id="child-frame" src="dummy.gif" style="display: none;"></iframe>
        </body>
</html>
```

代码清单 11.18　通过 iframe 实现跨源通信（子 iframe）

```
<html>
    <head>
        <title>子 iframe</title>
        <script>
        function executeApi() {
            // 将 location.hash 的第一个字符（#）去除，将剩余部分以 JSON 格式进行分析
            var param = JSON.parse(location.hash.substring(1));
            var xhr = new XHMHttpRequest();
            xhr.onreadystatechange = function() {
                if (xhr.readyState == 4 && xhr.status == 200) {
                    var iframe = document.getElementById('grandchild-iframe');
                    iframe.location.href = param.callback + '#' + xhr.responseText;
                }
            };
            xhr.open(param.api, 'GET');
            xhr.send(null);
        }
        </script>
    </head>
    <body onload='executeApi()'>
        <iframe id="grandchild-iframe" src="dummy.gif" style="display: none;"></iframe>
    </body>
</html>
```

代码清单 11.19　通过 iframe 实现跨源通信（孙 iframe）

```
<html>
    <head>
        <title>孙 iframe</title>
        <script>
        window.onload = function() {
            window.top.callback(location.hash);
        }
        </script>
    </head>
    <body></body>
</html>
```

尽管通过 iframe 实现的跨源通信比较复杂，但却具有在 Internet Explorer 中也能够正常工作的优点，而且也比通过 JSONP 实现的方式更为安全。JSONP 方式是无法对服务器端含有恶意代码的情况进行防范的，

而对于通过 iframe 的实现方式，由于只能由同一个域中的孙 iframe 对父页面进行操作，因此会更加安全。

11.2.11 window.postMessage

可以通过在 HTML5 中定义的 window.postMessage 来实现安全的跨源通信（代码清单 11.20、代码清单 11.21）。

代码清单 11.20 通过 postMessage 实现跨源通信（父页面）

```
<html>
    <head>
        <title>父页面</title>
        <script>
            // 在跨源通信中用于获取数据的函数
            function getData() {
                // 对子 iframe 进行 postMessage 操作
                frames[0].postMessage('http://other.example.com/some-data',
'http://other.example.com');
            }
            // 在跨源通信中作为回调函数被执行的函数
            // 被设定为用于接收来自子 iframe 的消息
            window.addEventListener('message', function(event) {
                if (event.origin !== 'http://other.example.com') {
                    return;
                }
                // 将结果保存于 event.data 中
                document.getElementById("result").innerHTML = event.data;
            }, false);
            </script>
        </head>
        <body>
            <input type="button" value="从 other.example.com 获取数据" onclick="getData()">
            <div id="result"></div>
            // 将 other.example.com 的页面指定为子 iframe 的 URL
            <iframe id="child-frame" src="http://other.example.com/api.html"
                style="display: none;"></iframe>
    </body>
</html>
```

代码清单 11.21 通过 postMessage 实现跨源通信（子 iframe）

```
<html>
    <head>
        <title>子 iframe</title>
        <script>
        window.addEventListener('message', function(event) {
            if (event.origin !== 'my.example.com') {
                return;
            }
            var xhr = new XHMHttpRequest();
            xhr.onreadystatechange = function() {
                if (xhr.readyState == 4 && xhr.status == 200) {
                    // 将 responseText 作为消息返回
                    event.source.postMessage(xhr.responseText, 'http://my.example.com');
                }
            };
            var url = event.data;
            xhr.open(url, 'GET');
            xhr.send(null);
        }, false);
        </script>
    </head>
    <body></body>
</html>
```

11.2.12　XMLHttpRequest Level 2

之前提到过，XMLHttpRequest 无法用于跨源通信。不过，这只是对于 Level 1 而言的。在 XMLHttpRequest Level 2 中，新增了一些功能以实现对跨源通信的支持。不过，要进行跨源通信就必须得到服务器端的许可。所以必须在响应中包含 Access-Control-Allow-Origin 这一 HTTP 头部，以指定可以访问的源。如果 Access-Control-Allow-Origin 这一头部的值被指定为了 "*" 的话，则表示允许来自任意源的访问。

在通过 XMLHttpRequest 进行跨源通信时，默认为不发送 Cookie。如果要发送 Cookie，则必须将 withCredentials 属性设置为 true（代码清单 11.22）。

代码清单 11.22　XMLHttpRequest Level 2

```
var xhr = new XMLHttpRequest();
xhr.open('GET', 'http://other.example.com', true);
xhr.withCredentials = true;    // 设定为将会发送 Cookie
xhr.onreadystatechange = function() {
    // 进行一些操作
};
xhr.send();
```

11.2.13　跨源通信的安全问题

通过跨源通信在不同的域中获取数据的做法已经被认为是理所应当的了，得到了广泛的使用。然而，由同源策略的制定而不难想到，在不同的域之间进行通信是存在安全风险的。需要对此时刻牢记。

应该只与可以信赖的域进行通信自不必说，另外，一个域即使在过去是安全的，也不能保证它永远就会是安全的。如果所要搭建的 Web 站点将会对个人信息进行处理，则更应该对安全性加以高度的重视。

11.3　表单

表单主要被用于用户注册等一些需要将数据发送给服务器以执行注册操作的处理之中。在用户注册这样的需要将各种信息（用户 ID、密码、邮件地址等）汇总发送至服务器的处理中，表单是一种简单而正确的 HTML 实现方法。

不过，在使用表单时也会有一些限制。为了避免这些问题，在有些情况下不会选择使用表单。

表单最大的不足就是在 submit 时会发生页面跳转。而 AJAX 却是一种不进行页面跳转，直接改写页面内容的技术。表单在 submit 时一定会发生页面跳转，因此在这一点上它是与 AJAX 的理念相悖的。在后面将会说到，其实还是有办法让表单在 submit 时不发生页面跳转的。不过在使用这种方法之后，是否还有必要使用表单倒是一个问题。我们来仔细考虑一下吧。例如，在 Twitter 中发送新的推特状态时用的并不是表单。这一过程中仅仅是通过 XMLHttpRequest 来进行了数据的发送而已。

得益于 AJAX，出现了不使用表单就能与服务器进行数据收发的方式。不过这也并不表示表单将会消失。在这里，我们将说明一下通过使用 JavaScript 来发挥表单功能的方法。这并不仅仅是为了向服务器发送数据，其目的是实现更为用户友好的表单功能。

11.3.1　表单元素

表单元素具有 HTMLFormElement 这一继承于 HTMLElement 的接口。表 11.3 总结了 HTMLFormElement 接口所定义的属性。

表 11.3 HTMLFormElement 的属性

属性名	说明
elements	form 内的 input 元素一览
length	form 内的 input 元素的数量
name	form 的名称。JavaScript 可通过它引用表单
acceptCharset	form 所支持的字符集
action	form 的 action 元素
enctype	form 的 content type
method	在发送数据时所用的 HTTP 类型
target	action 结果的写入目标
submit()	发送数据
reset()	将 form 还原至初始化状态

在向服务器 POST 数据时，可以通过 acceptCharset、action、enctype 与 method 来设定元信息。这些值都是能够被改写的。因此，我们甚至能够根据不同的按键操作而向其他 URL 发送请求。

elements 是对表单内的控件的引用。之后将会对表单控件进行详细说明。各个表单控件将按照它们在 HTML 中的书写顺序被引用。此外还能够通过 document.forms 对表单元素进行引用。这种引用同样会与它们在 HTML 中的书写顺序相一致（代码清单 11.23）。

代码清单 11.23 表单的引用

```
<body>
    <form>
        <input type="text">
        <input type="password">
        <input type="email"><!-- 希望获取该元素的值 -->
    </form>
    <script>
        var email = document.forms[0].elements[2].value;
        alert(email);
    </script>
</body>
```

然而，在处理表单时会受到 HTML 中元素书写顺序的影响，这并不是什么好事。即使只是对 HTML 的书写进行小幅修改，也有可能导致程序无法正常运行。为此，可以使用 <form> 标签或 <input> 标签中的 name 属性，通过这一 name 属性的值来引用元素。这样一来，只要这些名称不发生变化，脚本的功能就不会受到影响（代码清单 11.24）。

代码清单 11.24 通过名称引用表单

```
<form name="user">
    <input name="username" type="text">
    <input name="password" type="password">
    <input name="email" type="email"><!-- 希望获取该元素的值 -->
</form>
<script>
    var email = document.user.email.value;
    alert(email);
</script>
```

submit() 方法被执行后的效果与按下 submit 键的效果一样，都将会向服务器发送数据。不过，它与按下 submit 键时不同的是，submit 事件不会被触发。于是，onsubmit 事件处理程序也不会被执行。reset() 方法与 reset 键在这一点上也是如此。只有在按下 reset 键时 onreset 事件处理程序才会被执行，而如果是调用 reset() 方法则不会执行。

如果 onsubmit 事件处理程序返回了 false 的话，表单的数据就不会被发送至服务器。之后，本章还将对“内容验证”进行说明。只要在 onsubmit 事件处理程序中执行对内容的验证，并在发现内容不完整时返回 false，就能够避免发送没有意义的数据。同样地，如果 onreset 事件处理程序返回了 false，则表单将

不会重置。

11.3.2　表单控件

在表单中用于接收输入信息的元素称为表单控件。常用的表单控件有 input 元素、select 元素、button 元素与 textarea 元素等。一方面这些元素都具有不同的接口，但另一方面他们都包含一些通用的属性。表 11.4 总结了这些表单控件中的通用属性。

表 11.4　表单控件的通用属性

属性	说明
form	该控件所属的表单元素
disabled	控件是否被禁用
name	控件的名称
type	控件的
value	控件的值
focus()	使控件获得焦点（*1）
blur()	使控件失去焦点（*1）

*1 在 button 元素中不含这一属性

如果 disable 属性的值为 true，该表单控件就会被禁用，无法进行输入操作。通过对 disable 属性设置恰当的值，就能够控制输入，实现例如仅能够在满足了特定的条件时输入这样的效果。

通过以 JavaScript 调用 focus() 方法与 blur() 方法，可以使特定的元素获得或失去焦点。与 submit() 和 reset() 方法不同的是，在执行这些方法的时候将会分别触发 focus 事件与 blur 事件。

11.3.3　内容验证

■ 内容验证的必要性

在输入时，对于必须输入的项目是否已经填入了所需的值，或者是否超过了允许输入的文字数量等情况，我们常常会通过 JavaScript 来检查。虽说最终自然还是应该由服务器端对其检查，不过在客户端也需要执行检查工作。这样做有很多好处。

首先想到的一个好处是，由于在客户端就完成了检查工作，在与服务器通信过程中的时间损失将会消除。与将数据发送给服务器，由服务器对数据进行检查，并在收到服务器返回的数据出错的响应之后再进行画面绘制的方式相比，在客户端检查数据以判断是否正确并绘制画面的方式性能更好。

此外，这种做法可以避免向服务器发送明显有误的数据，从而减轻了服务器的处理量。

不过，不能忘记最后应该由服务器端检查数据。这是由于 JavaScript 是在客户端执行操作的，可以对数据进行各种各样的改写，仅靠这种方式并不能确保数据的完整性。

■ 进行内容验证的时机

有多次进行内容验证的时机。其中之一自然是在按下 submit 键时验证。在按下了 submit 键之后，检查整个 form 内所有的元素的值，如果发现了不合法的数据，则返回 false，并取消数据的发送。

或者也可以在数据输入之后立即检查该数据。如果对能够输入的字符数进行了限制，则可以在输入时逐一对字符计数，如果字符数超过了限制，则改变背景色以向用户提供反馈。或者在输入用户 ID 等不允许重复的数据时，在输入后与服务器进行通信，检查该用户 ID 是否已经存在。

这一类的检查时的做法与其他的 JavaScript 处理并没有太大的区别。只需要将事件与处理相关联，并反馈处理结果即可。input 元素可以触发各种各样的事件，因而可以实现周密的行为控制。

11.3.4 可用于验证的事件

■ submit

在使用表单时最为重要的一个事件就是 submit 事件。不过正如之前所讲，在调用 submit() 方法时并不会触发该事件。

仅通过 submit 事件的事件处理程序就能够取消表单的发送操作。此外，由于这是最后一个能够进行内容验证的事件，因此即使已经在之前通过各元素自身的事件对内容进行了验证，也可以在 submit 事件被触发时再一次检查所有元素。

■ focus、blur

input 元素在获取了焦点时将会触发 focus 事件。而当其失去焦点时，则将触发 blur 事件。

在 focus 事件被触发后，通过更改 input 元素的背景色等视觉上的方式，帮助用户了解正在对哪一个 input 进行操作，是一种不错的做法。而对于 blur 事件来说，则可以反过来执行将背景色还原的操作。不过在最近的浏览器中，有很多都会对正处于焦点的元素进行强调显示，所以也不是非要进行这样的处理。

可以认为元素在触发了 blur 事件时相应的输入已经完成，所以在这时验证内容也是一种妥当的选择。不过在这种情况下，焦点已经转移至了下一个输入元素，所以应该怎样提供反馈以增强用户体验是一个问题。

■ change

在 input 元素的值发生变化时该事件将被触发。虽然文本框也对这一事件提供了支持，不过它主要还是被用于复选框或单选框等用于选择值择的 input 元素之中。

之所以说无法在文本框中使用该事件，是因为文本框中 change 事件的触发时机很难被有效利用。对于文本框来说，change 事件仅会在文本框失去焦点，且其 value 属性的值与具有焦点时不同的情况下，才会被触发。因此，在输入过程中该事件不会被触发。change 事件本应该是在有字符输入而改变了 value 时被触发，而在文本框中无法实现这一效果，故而无法对其使用。

■ keydown、keyup、keypress

keydown、keyup 与 keypress 事件将在发生键盘输入时被触发。当有键被按下时将触发 keydown，当释放按下的键时将触发 keyup，当有键被按下且输入了字符时将触发 keypress。不过，与键盘相关的事件的执行方式会根据浏览器以及系统平台的不同而有所差异，所以很难对其进行处理。甚至，keydown/keyup 事件与 keypress 事件所取得的 keyCode/charCode 也不相同。

■ input

当 input 元素发生了输入行为时将会触发 input 事件。对于文本框来说，每输入 1 个字符都会触发该事件。而这正是之前期望通过 change 事件所获得的效果。不过，input 事件是在 HTML5 的标准中被定义的，所以在 Internet Explorer 8 以及更早的版本中不被支持。

input 事件与 keypress 事件等由键盘输入触发的事件不同，只有在值发生变化时才会被触发。也就是说，即使通过方向键改变了光标插入标记的位置，也不会触发 input 事件。此外，keypress 事件在通过退格键删除已输入的字符时不会被触发，而与之相对的，input 事件在进行删除时也会被触发。

11.3.5 使用表单而不产生页面跳转的方法

在 form 元素中含有一个 target 属性。form 将会在这一 target 属性所指定的框架或窗口中绘制 submit 结果的响应。如果没有指定 target 属性的值，则该值默认是自身所属的框架或窗口，这时将会发生页面

跳转。表 11.5 总结了 target 属性可以设定的值。

表 11.5　form 的 target 属性

值	显示结果的位置
_blank	新建窗口
_self	当前框架（窗口）
_parent	父框架
_top	解除框架的分割并在整个窗口中显示
框架名、窗口名	所指定的任意框架（窗口）

为了防止页面跳转的发生，可以将 target 指定为当前窗口之外的窗口。不过打开新窗口的做法会有些碍眼，甚至可能会被浏览器的弹窗锁定功能所拦截。因此应该将 target 指定为空的 iframe，并将该 iframe 的宽度和高度都设为 0，使其不显示。

可以让对 parent 窗口进行操作的 JavaScript 来对结果页面进行处理。这样一来，就能够在不发生页面跳转的情况下利用 form 了（代码清单 11.25）。

代码清单 11.25　使用了表单的通信

```
<script>
    // 该函数将被 submit 了 form 的结果页面进行调用
    function onComplete() {
        alert('complete.');
    }
</script>
<!-- 将名为 result 的 frame 指定为发送结果的写入目标 -->
<form target="result" action="register">
    <input type="text">
    <input type="submit" value="post">
</form>
<!-- 写入发送结果的 iframe -->
<iframe name="result" style="width: 0; height: 0; border: none;"></iframe>

<!-- 结果页面 -->
<!DOCTYPE HTML>
<html lang="zh-CN">
<head>
    <meta charset="UTF-8">
    <script>
        // 执行 parent 窗口的 onComplete 函数
        parent.onComplete();
    </script>
</head>
<body>
</body>
</html>
```

不过，无法否认的是，使用这样的方式还是会有种破解页面的感觉。

第 12 章 库

使用了库，就可以不必书写那些处理约定俗成的代码，也不用考虑跨浏览器支持等烦杂内容，从而能够简单地实现这些功能。本章将以因链式语法及插件系统而广受好评的 jQuery 为中心，讲解库的使用方法。

12.1 使用库的原因

在客户端 JavaScript 中，最为费时费力的事就是跨浏览器支持，更进一步说，是对 Internet Explorer 的支持。Internet Explorer 6、7、8 之间都有着细微的功能差异，而且从浏览器市场份额的角度来看，这真是一个不可忽视的麻烦问题。当然了，很多大型网站已经不再支持 Internet Explorer 6，所以忽略它可能也不会有什么问题。例如，Yahoo! JAPAN 已经不支持 Internet Explorer 6，Google 则将 Google Apps 所支持的浏览器限定为最新版的 Google Chrome、Firefox、Safari、Internet Explorer 及其上一个版本。

然而，即使不考虑 Internet Explorer 6，仍然无法逃脱 Internet Explorer 的束缚。要让每一个开发者各自解决这一问题是不可行的，应该通过已经被很多网站所使用的库来尽可能地减少花费在跨浏览器支持上的时间与精力。话虽如此，库也不是完美无缺的。对于库无法涵盖的部分，仍需要自己书写跨浏览器支持的代码。在充分使用了库的基础上，理解浏览器的功能，在不得已的情况下自己写出相应的处理，这一点很重要。

如果要开发真正的 Web 应用，就必定需要使用库。不过，如果仅仅知道库的使用方法，发生问题时还是无法解决。因此，至少还应该掌握之前所说的那些知识，并在此基础上，理解使用库所能带来的优点并对其恰当地加以利用。

12.2 jQuery 的特征

jQuery 是现在世界上使用最多的 JavaScript 库，其作者为 John Resig，可以通过下面的 URL 访问 jQuery 的官方站点。在执笔本书时的最新版为 1.6.2。jQuery 是通过 MIT 许可进行发布的。

http://jquery.com/

jQuery 具有以下特征。

- 压缩后仅有 31KB，非常轻巧
- 通过链式语法实现
- 通过 CSS3 选择器及自定义选择器获取元素
- 支持插件，可扩展性高

jQuery 并不包含 UI 组件。jQuery 另有一个独立的 UI 组件，名为 jQuery UI。jQuery UI 的官方站点如下所示。在本书中将会对 jQuery 进行说明，而不会涉及 jQuery UI。只需要了解 jQuery，就足以能掌握 jQuery 的基本使用方法了。

http://jqueryui.com/

12.3 jQuery 的基本概念

12.3.1 使用实例

在使用 jQuery 时，由于其采用了自定义的选择器以及链式语法，因此看起来像是在书写不同于 JavaScript 的另一种语言。例如，点击 foo 类中的第一个 div 元素时，创建一个链接并将其添加的操作，可以像代码清单 12.1 这样来书写。

代码清单 12.1　jQuery 的使用实例

```
<button class='foo'>当点击该按钮时，添加一个链接</button>
<button class='foo'>该按钮不会有反应</button>
<script>
// 对点击 class="foo" 的第一个 button 元素时的处理进行设定
$('button.foo:first').click(function() {
    $('<div><a></a></div>')              // 创建 div 元素。div 元素中 a 元素这一子元素
                                         // 这时选择的是 div 元素
    .find('a')                           // 对 a 元素进行选择。所选择的目标从 div 元素转为了 a 元素
        .text('jQuery.com')              // 将 a 元素的文本设定为 jQuery.com
        .attr('href','http://jquery.com') // 将 a 元素的 href 属性设定为 http://jquery.com
    .end()                               // 结束对 a 元素的选择
                                         // 在 a 元素被选择之前所选中的 div 元素回到了被选择的状态
    .appendTo('body');                   // 将 div 元素添加至 body
});
</script>
```

尽管上面例子中的功能并没有什么特别意义，不过也能从中看出借助 jQuery 就能够实现简洁的书写。在某个元素触发了事件之后对另一个元素进行操作是在 JavaScript 中常见的处理，而对此 jQuery 只需要书写这么一点的代码就能实现。完全不需要使用 getElementById() 或 firstChild 之类的 DOM API。可以说 jQuery 是尽可能地隐藏了 DOM API 并对其进行了重新定义。

作为比较，在代码清单 12.2 中没有使用 jQuery，而是通过标准的 JavaScript 与 DOM API 来实现上一个例子中的功能。

代码清单 12.2　不使用 jQuery 的例子

```
<button class='foo'>当该按钮被点击时，添加一个链接</button>
<button class='foo'>该按钮不会有反应</button>
<script>
// 在被点击时所进行的处理
var onClick = function() {
    var div = document.createElement('div');
    var a = document.createElement('a');
    a.appendChild(document.createTextNode('jQuery.com'));
    a.href = 'http://jquery.com';
    div.appendChild(a);
    document.body.appendChild(div);
};

// 通过 getElementsByTagName() 来获取所有的 button 元素
var buttons = document.getElementsByTagName('button');
// 在所取得的 button 元素中找到第一个类名包含了 foo 的元素
// 对点击事件设定事件侦听器
for (var i = 0; len = buttons.length; i < len; i++) {
    if (buttons[i].className.match(/(^|\s)foo(\s|$)/)) {
        buttons[i].addEventListener('click', onClick, false);
        break;
```

```
        }
    }
</script>
```

或许仅仅是这种程度的功能还不至于使书写变得过于复杂，但代码量几乎是倍增。在使用 jQuery 时无需用到的变量、DOM API、if 语句以及 for 语句等控制语句的书写使得代码量增加。而变量与控制语句的使用也会增加出现错误的可能，所以如果能够减少使用的话，还是少用一些为好。

12.3.2 链式语法

■ 链式语法的定义

在之前的例子（代码清单 12.1）中，事件侦听器的描述其实只用了 1 行，分号只是出现在最后的 appendTo() 方法的结尾处而已。在此之前全都是通过点运算符将方法连接起来的。为什么能够使用这样的书写方式呢？

jQuery 对象中的大部分方法都会返回一个 jQuery 对象。因此，如果执行了一个返回 jQuery 对象的方法，就能够对所返回的值再一次执行能返回 jQuery 对象的方法。以这种方式将方法连在一起书写的方式称为链式语法。

■ 链式语法中的 jQuery 对象

需要注意的是，执行方法的 jQuery 对象与方法所返回的 jQuery 对象并不一定总是同一个对象。有时候会在方法内创建一个新的 jQuery 对象并将其返回。因此，如果只是简单地将链式方法拆开分行书写的话，并不一定能获得所期望的结果（代码清单 12.3）。为了使其正常执行功能，我们必须将目标对象替换为恰当的 jQuery 对象（代码清单 12.4）。

可以通过在 jQuery 中更改所返回的 jQuery 对象来实现链式语法中操作对象的替换。在执行 find() 方法这样的用于对元素进行选择的方法时，可以替换目标元素集。而在执行了 end() 方法之后，所选中的元素集就会回到之前一次时的状态。通过使用 end() 方法，不但可以将目标元素集替换为恰当的值，还能够使链式语法连得很长。不过，为了准确地运用 end() 方法，必须对之前的状态有着清楚的了解。否则就无法知道 end() 方法的返回值是以哪一个元素为目标的，从而无法执行期望的操作。以链式语法连起来书写的方式确实很方便，不过在书写时必须正确理解正在操作的对象才行。

代码清单 12.3　没有使用链式语法的不正确的书写方式

```
// 不使用代码清单 12.1 中的链式语法来描述事件侦听器
// 仅仅简单地对已有的对象依次执行方法并不能得到所期望的执行结果
var elem = $('<div><a></a></div>');
elem.find('a');                          // 对 a 元素进行选择并不会将 elem 的引用替换为 a 元素
elem.text('jQuery.com');                 // 对 div 元素进行的操作
elem.attr('href','http://jquery.com');   // 对 div 元素进行的操作。无法生成链接
elem.end();
elem.appendTo('body');
```

代码清单 12.4　没有使用链式语法的正确的书写方式

```
// 不使用代码清单 12.1 中的链式语法来描述事件侦听器
// 由于同时以 div 与 a 这两个元素为目标进行处理，因此执行结果与所期待的一致
var div = $('<div><a></a></div>');
var a = div.find('a');
a.text('jQuery.com');
a.attr('href','http://jquery.com');
// 如果没有使用链式语法，a.end() 就没有意义了，所以不需要写出
div.appendTo('body');
```

专栏

链式语法的缺点

JavaScript 的主要功能之一为对 DOM 树中的元素进行选取与操作。在使用了 jQuery 之后，就能以非常简洁的书写方式实现对这一系列的处理的表述。不过，这种方便的链式语法也有其缺点。它存在调试时无法对其设定断点的问题。

开发者无法在链式语法连接起来的代码段中的特定位置设定断点。如果所写的代码没有复杂到不得不在调试时设定断点才能跟踪处理过程的话倒还好，但万一特别复杂的话，情况就会变得很麻烦。不过，最近出现了一些进化幅度令人惊讶的 JavaScript 调试器，如有可能，希望大家能尝试使用一下。

12.4　$ 函数

通过前面的例子我们可以知道，在 jQuery 中 $ 函数（jQuery 函数）根据参数的不同可以执行各种操作。jQuery 的 $ 函数将会根据当前的上下文语境选择最合适的操作。下面总结了 $ 函数的功能。

12.4.1　抽取与选择器相匹配的元素

在将 CSS 选择器传递给了 $ 函数之后，就能够抽取出与其相匹配的元素。而通过第 2 个参数还能够制定搜索范围。即使浏览器不支持在 CSS2 或 CSS3 中引入的选择器，也能够使用这一功能。

```
// 通过 CSS 选择器，从 id="foo" 的元素的子元素中抽取出 class="bar" 的 div 元素
$('#foo div.bar');

// 同样是从 id="foo" 的元素的子元素中抽取出 class="bar" 的 div 元素
$('div.bar', '#foo');

// 以下面的方式指定参数也能获得相同的结果
var foo = document.getElementById('foo');
$('div.bar', foo);
```

另外，除了 CSS 选择器之外还能够使用 jQuery 自定义的选择器。将在之后说明能够在选择器中使用的语法。

12.4.2　创建新的 DOM 元素

如果将可以被解释为 html 标签的字符串传递给 $ 函数，则能够创建新的元素。

$('<div> 新的 div 元素 </div>');

12.4.3　将已有的 DOM 元素转换为 jQuery 对象

如果将已有的 DOM 元素传递给 $ 函数，就能够将其转换为 jQuery 对象。

```
// 将 body 元素转换为 jQuery 对象
$(document.body);
```

12.4.4　对 DOM 构造完成后的事件侦听器进行设定

如果把一个 Function 对象传递给 $ 函数，则能让该函数在 DOM 构造完成后执行。这与对 document 的 ready 事件设定事件侦听器的写法的效果是等价的。之后还会详细说明。

```
$(function() {
// DOM 构造完成后的处理
});

// 等同于这种方式
$(document).ready(function() {
// DOM 构造完成后的处理
});
```

12.5 通过 jQuery 进行 DOM 操作

12.5.1 元素的选择

可以通过 $ 函数选择元素。表 12.1 总结了包括 CSS 选择器在内的 jQuery 所能使用的选择器。对于 jQuery 自定义的选择器，将用○进行标识。

表 12.1 jQuery 所能使用的选择器

选择器语法	自定义	被选择的元素
*	–	所有的元素
#id	–	指定 id 的元素
.class	–	指定 class 的元素
tag	–	标签名为 tag 的元素
选择器 1, 选择器 2, 选择器 N	–	与指定选择器中的某个相匹配的元素
parent > child	–	与选择器 parent 相匹配的元素的直接子元素中与选择器 child 相匹配的元素
ancestor descendant	–	与选择器 ancestor 相匹配的元素的子元素中与选择器 descendant 相匹配的元素
prev + next	–	与选择器 prev 相匹配的元素的下一个兄弟元素，且还需要与选择器 next 相匹配
prev ~ siblings	–	与选择器 prev 相匹配的元素之后的兄弟元素，且还需要与选择器 siblings 相匹配
[attr]	–	具有属性 attr 的元素
[attr="val"]	–	属性 attr 的值为 val 的元素
[attr!="val"]	○	属性 attr 的值不为 val 的元素
[attr^="val"]	–	属性 attr 的值以 val 开始的元素
[attr$="val"]	–	属性 attr 的值以 val 结束的元素
[attr*="val"]	–	属性 attr 的值含有 val 的元素
[attr~="val"]	–	将属性 attr 的值以空格进行分割后，含有值为 val 的片段的元素
[attr|="val"]	–	属性 attr 的值为 val 或以 val– 开始的元素 (*1)
[属性选择器 1][属性选择器 2][属性选择器 N]	–	与指定的多个属性选择器都匹配的元素
:contains(text)	–	在文本内容中包含 text 的元素
:empty	–	没有子元素（包括文本）的元素
:parent	○	拥有子元素（包括文本）的元素
:has(sel)	○	拥有与选择器 sel 相匹配的子元素的元素
:header	○	h1 ～ h6 元素
:animated	○	动画中的元素
:not(sel)	–	与选择器 sel 不匹配的元素
:first	○	第一个的元素
:last	○	最后一个的元素
:even	○	第偶数个元素
:odd	○	第奇数个元素
:eq(n)	○	第 n 个元素
:gt(n)	○	第 n 个元素之后的元素（不包括第 n 个元素）
:lt(n)	○	第 n 个元素之前的元素（不包括第 n 个元素）
:first–child	–	第一个子元素
:last–child	–	最后一个子元素
:nth–child(n)	–	第 n 个子元素

（续）

选择器语法	自定义	被选择的元素
:only-child	–	没有兄弟元素的元素
:hidden	○	隐藏的元素 (*2)
:visible	○	可见元素 (*2)
:focus	–	处于焦点的元素
:disabled	–	处于禁用状态的元素
:enabled	–	处于启用状态的元素
:checked	○	勾选的元素
:selected	○	option 元素中被选择的元素
:input	○	input、textarea、select、button 元素
:checkbox	○	type="checkbox" 的元素
:radio	○	type="radio" 的元素
:file	○	type="file" 的元素
:image	○	type="image" 的元素
:text	○	type="text" 的元素
:password	○	type="password" 的元素
:button	○	button 元素或 type="button" 的元素
:submit	○	type="submit" 的元素
:reset	○	type="reset" 的元素

*1 主要用于选择语言代码。例如，可以通过 a[hreflang|="ja"] 的方式来选择日语的页面链接。这时，hreflang 为 ja 或 ja-JP 的元素都能够被选择。

*2 如果与以下条件中的某些相符，元素则会被认为是隐藏。

- 在 CSS 中被指定为 "display: none" 的元素
- type="hidden" 的 input 元素
- 宽与高都为 0 的元素
- 父元素是隐藏元素

而在 CSS 中被指定为 "visibility: hidden" 或 "opatcity:0" 的元素不会被判定为隐藏元素

此外，还可以以当前选中的元素集为基础，进一步进行筛选，或以所指定的相对关系选择元素。表 12.2 总结了与此相关的方法。

表 12.2　用于选择元素的方法

方法	说明
find(sel)	在所有的子元素中选择与选择器 sel 相匹配的元素
contents()	选择所有含有文本节点的直接子元素
children([sel])	选择直接子元素。可以通过选择器 sel 过滤
sibling([sel])	选择所有的兄弟元素。可以通过选择器 sel 过滤
next([sel])	获取紧接着的下一个元素。可以通过选择器 sel 过滤
nextAll([sel])	选择之后所有的兄弟元素。可以通过选择器 sel 过滤
nextUntil([until,][sel])	选择之后跟着的所有兄弟元素。如果指定了选择器 until，则将只选择到与其相匹配的元素为止。可以通过选择器 sel 过滤
prev([sel])	获取相邻的前一个元素。可以通过选择器 sel 过滤
prevAll([sel])	选择之前所有的兄弟元素。可以通过选择器 sel 过滤
prevUntil([sel])	选择之前所有的兄弟元素。如果指定了选择器 until,则将只选择到与其相匹配的元素为止。可以通过选择器 sel 过滤
parent([sel])	选择直接父元素。可以通过选择器 sel 过滤
parents([sel])	选择所有的祖先元素。可以通过选择器 sel 过滤
parentsUntil([until,][sel])	选择所有的祖先元素。如果指定了选择器 until,则将只选择到与其相匹配的祖先元素。如果指定了选择器 sel，则将只选择与之相匹配的元素
offsetParent()	选择距所指定的 position 最近的父元素
closest(sel, [context])	以自己为起点向上遍历 DOM 树，选择第一个与选择器 sel 小相匹配的元素。自己本身也是可被选择的对象。如果指定了元素 context，则将以此为过滤条件，只选择存在于该元素内的元素
filter(sel)	选择与选择器 sel 相匹配的元素。如果 sel 被指定为一个函数，则将选择能使该函数返回 true 值的元素 (*1)
not(sel)	选择与选择器 sel 相匹配的元素。如果 sel 被指定为一个函数,则将选择能使该函数返回 false 值的元素 (*1)
eq(n)	选择第 n 个元素。如果 n 被指定为了负值，则选择倒数第 n 个元素

（续）

方法	说明
has(sel)	选择含有与选择器 sel 相匹配的子元素的元素
first()	选择第一个元素
last()	选择最后一个元素
slice(start, [end])	选择从第 start 个起至第 end 个的元素。如果没有指定 end，则将选择到元素集的最后。如果设定了负值，则将从元素集的最后开始向前倒数
is(sel)	对是否匹配选择器 sel 进行判断。返回值是 Boolean 型
map(func(n, elem))	通过回调函数 func 来返回对各个元素的处理结果。回调函数的参数是元素的 index 与 DOM 元素
add(sel)	向当前元素集中添加与选择器 sel 相匹配的元素集
andSelf()	向当前元素集中添加之前一个状态的元素集
end()	将元素集的选择状态还原为之前一个状态

(*1) 元素的 index 将被作为参数传递给函数。而函数内的 this 所引用的就是该 DOM 元素。

12.5.2　元素的创建・添加・替换・删除

可以通过 $ 函数来创建元素。另外，还可以通过 append() 方法同时执行元素的创建与添加。如果要进行替换操作，可以使用 replaceWith() 等方法，如果要进行删除操作，则可以使用 remove() 等方法。表 12.3 总结了与元素操作相关的方法。

表 12.3　用于对元素进行操作的方法

方法	说明
append(content, [content])	将 content 添加到子元素的最后
append(func(index, htmlStr))	将函数 func 的结果添加到子元素的最后 函数将接收元素的下标与 HTML 字符串，并返回一个 HTML 字符串
appendTo(target)	将当前的元素集添加到 target 子元素的最后
prepend(content, [content])	将 content 添加到子元素的最后
prepend(func(index, htmlStr))	将函数 func 的结果添加到子元素的头部 函数将接收元素的下标与 HTML 字符串，并返回一个 HTML 字符串
prependTo(target)	将当前的元素集添加到 target 子元素的头部
html(htmlStr)	以 HTML 字符串 htmlStr 创建 DOM 并设为元素
html(func(index, htmlStr))	将函数 func 的结果设为元素。 函数将接收元素的下标与 HTML 字符串，并返回一个 HTML 字符串
text(str)	将字符串 str 设为元素
text(func(index, str))	将函数 func 的结果设为元素。函数将接收元素的下标与字符串，并返回一个字符串
after(content, [content])	将 content 添加到元素之后
after(func(index))	将函数 func 的结果添加到元素之后。函数将接收元素的下标，并返回一个 HTML 字符串
before(content, [content])	将 content 添加到元素之前
before(func(index))	将函数 func 的结果添加到元素之前。函数将接收元素的下标，并返回一个 HTML 字符串
insertAfter(target)	将元素添加到 target 之后
insertBefore(target)	将元素添加到 target 之前
replaceWith(content)	将元素替换为 content
replaceWith(func)	将元素替换为函数 func 的返回值。函数返回的是 HTML 字符串
replaceAll(target)	将 target 全部替换为当前元素
wrap(content)	以 content 来包裹元素
wrap(func(index))	以函数 func 的返回值来包裹元素。 函数将接收元素的下标，并返回一个 HTML 或 jQuery 对象
wrapInner(content)	以 content 来包裹元素中的内容
wrapInner(func(index))	以函数 func 的返回值来包括元素中的内容。 函数将接收元素的下标，并返回一个 HTML 或 jQuery 对象
wrapAll(content)	以 content 来包裹整个元素集
unwrap()	删除元素外所包裹的父元素
remove([sel])	删除元素集中与选择器 sel 相匹配的元素。 如果没有指定选择器，则删除所有内容
detach([sel])	与 remove() 相同。不过将会保留事件侦听器的设定

（续）

方法	说明
empty()	将元素清空
clone([withDataAndEvents,] [deepDataAndEvents	复制元素。如果第 1 个参数为 true（默认为 true）则会一起复制事件侦听器。如果第 2 个参数与 true（默认为 true），则会一起复制子元素

12.6 通过 jQuery 处理事件

12.6.1 事件侦听器的注册・删除

■ bind()/unbind()

在 jQuery 中，我们可以通过 bind() 方法来注册事件侦听器。需要向该方法传递的是事件类型与事件侦听器。也可以传递事件类型与事件侦听器的映射，以同时注册多个事件侦听器。

而如果要删除事件侦听器，则可以使用 unbind() 方法。需要向该方法的参数传递的是事件类型与函数。如果没有指定参数，则将删除元素中所有被设定的事件侦听器。在代码清单 12.5 中是 bind() 方法与 unbind() 方法的例子。

代码清单 12.5　通过 jQuery 注册事件侦听器

```
<div class='foo'>foo</div>
<div class='bar'>bar</div>
<div class='baz'>baz</div>
<script>
// 为 click 事件注册事件侦听器
$('.foo').bind('click', function() {
    $(this).text('Hello!');
});
// 同时注册多个事件侦听器
$('.bar').bind({
    'mouseover': function() {
        $(this).text('Hi!');
    },
    'mouseout': function() {
        $(this).text('Bye!');
    }
});
// 当点击 baz 时，将会删除 bar 的 mouseout 事件的事件侦听器
$('.bar').bind('click', function() {
    $('.bar').unbind('mouseout');
});
</script>
```

■ live()/die()

还可以通过 live() 方法来注册事件侦听器。其使用方式与 bind() 相同。

bind() 与 live() 的区别在于，通过 bind() 注册的事件侦听器只能对在执行 bind() 时已经存在的元素产生效果，而通过 live() 注册的事件侦听器则能对在执行了 bind() 之后才添加的元素也产生效果。之所以在事件侦听器被设定之后才添加的元素也能够受到事件侦听器的侦听，是因为 live() 是对 document 对象进行事件侦听器的设定的。对 document 对象设定的事件侦听器的处理流程是，首先检查冒泡阶段的事件，如果事件与 live() 所指定的选择器或事件类型相匹配，则将执行所注册的事件侦听器。

可以通过 die() 方法来解除通过 live() 设定的事件侦听器。这与 unbind() 的使用方法相同。

■ delegate()/undelegated()

之前提到过，在使用 live() 时，其实事件侦听器是被设定于 document 对象之上。而如果使用的是

delegate() 方法，则能够对 document 对象之外的对象执行 live() 以设定事件侦听器。也就是说，从功能上来说以下两行是相同的。

```
$('.foo').live('click', functio() { /* 处理 */ });
$(document).delegate('.foo', 'click', function() { /* 处理 */ });
```

虽说两者的功能是一样的，但 delegate() 的性能比 live() 更为优秀。这是因为 live() 在 $('.foo') 时会查找 foo 类的元素，而 delegate() 则不需要进行这样的查找操作。foo 类这一信息只被用于事件被触发时对该事件目标是否与条件相匹配进行判断，而不必在设定事件侦听器时对 foo 类的元素进行检索。因此，可以说不进行无用的元素检索的 delegate() 在性能上更为优秀。

如果要解除通过 delegate() 设定的事件侦听器，则可以使用 undelegate() 方法。

■ one()

one() 方法是一种 bind() 的特殊形式。通过 one() 注册的事件侦听器只会被执行一次。one() 的使用方式与 bind() 相同。

12.6.2 事件专用的事件侦听器注册方法

一些标准事件还提供了相应的专用方法，也可以使用这些方法来注册事件侦听器。

提供了专用方法的事件有以下这些。关于各个事件的意义，请参见 10.6.2 节。

- click
- dblclick
- mousedown
- mouseup
- mousemove
- mouseout
- mouseover
- mouseleave
- mouseenter
- keydown
- keypress
- keyup
- change
- blur
- focus
- focusin
- focusout
- select
- submit
- resize
- scroll
- load
- unload
- error

这些方法的第 1 个参数用于接收需要传递给事件侦听器的数据的映射（可以省略），而第 2 个参数则用于接收事件侦听器。可以通过 Event 对象的 data 属性来引用第 1 个参数所指定的映射。如果在不传递参数的情况下执行这些方法，将会触发相应的事件。代码清单 12.6 是其使用示例。

代码清单 12.6 事件专用的事件侦听器注册方法

```
<div class='foo'>Click Me!</div>
<div class='bar'>Double Click Me!</div>
<script>
// 对 click 事件注册事件侦听器
$('.foo').click({ message: 'Hello!' }, function (event) {
    alert(event.data.message); // => 将会显示 Hello! 这一提示
});
// 当双击 .bar 时，将会触发 .foo 的 click 事件
$('.bra').dblclick(function() {
    $('.foo').click();
});
</script>
```

12.6.3 ready() 方法

在 jQuery 中，我们可以像代码清单 12.7 这样，通过 ready() 方法对 DOM 构造完成后的处理进行设定。

代码清单 12.7　通过 jQuery 进行基本的 JavaScript 处理

```
$(document).ready(function() {
// 这里写的是需要在 document 对象的 ready 事件中执行的处理
// 也就是说，这里要写的是初始化处理（通常会对其他元素的事件处理程序进行设定）
// ready 事件的触发时机与 DOMContentLoaded 事件的触发时机相同
});

// 也可以使用像下面这样省略写法
$(function() {
});
```

12.7　通过 jQuery 对样式进行操作

jQuery 提供了一些方便的方法以对样式进行操作。从用于更改类名的方法到各种动画，都能够简单地进行操作。

12.7.1　基本的样式操作

jQuery 提供了一些能够执行更改元素所设定的类名或 CSS 属性等基本样式操作的方法。表 12.4 总结了这些方法。

表 12.4　基本的样式操作

方法	说明
css(prop)	获取 CSS 属性 prop 的值
css(prop, val)	将 CSS 属性 prop 的值设定为 val
css(prop, func(index, val))	将 CSS 属性 prop 的值设定为函数 func 的返回值。 函数将接收元素的下标与当前的 prop 值，并返回将要设定的值
css(props)	同时设定多个 CSS 属性。props 是属性名与属性值的映射
addClass(clazz)	将类 clazz 添加到元素
addClass(func(index, clazz))	将番薯 func 的返回值添加到类中。 函数将接收元素的下标与当前的类，并返回将要添加的类
removeClass([clazz])	从元素中删除类 clazz。如果没有指定参数，则将删除所有的类
removeClass(func(index, clazz))	从类中删除函数 func 的返回值。 函数将接收元素的下标与当前的类，并返回将要删除的类
toggleClass(clazz)	如果元素具有类 clazz，则将其删除，否则添加该类
toggleClass(func(index, clazz))	如果元素具有函数 func 的返回值，则将其删除，否则添加该返回值。函数将接收元素的下标与当前的类，并返回将要进行操作的类
hasClass(clazz)	判断元素是否含有类 clazz。返回值是 Boolean 型
height()	获取不含 border 及 padding 等的元素高度。单位为 px
innerHeight()	获取含有 padding 的元素高度。单位为 px
outerHeight([includeMargin])	获取含有 border 的元素高度。 如果参数被设定为 true，则获取同时包含 border 与 margin 的元素高度。单位为 px
width()	获取不含 border 及 padding 等的元素宽度。单位为 px
innerWidth()	获取含有 padding 的元素宽度。单位为 px
outerWidth([includeMargin])	获取含有 border 的元素宽度。 如果参数被设定为 true，则获取同时包含 border 与 margin 的元素宽度。单位为 px
height(val)	设定元素的高度。对于所传递的参数，将会作为像素值处理
height(func(index, h))	将函数 func 的返回值作为高度进行设定。 函数将接收元素的下标与当前的高度，并返回将要设定的高度。
width(val)	设定元素的宽度。对于所传递的参数，将会作为像素值处理
width(func(index, h))	将函数 func 的返回值作为宽度进行设定。 函数将接收元素的下标与当前的宽度，并返回将要设定的宽度。
offset()	获取元素在文档坐标中的位置 (*1)
position()	获取元素与以 position 属性所指定的第一个父元素之间的相对位置 (*1)

（续）

方法	说明
offset(pos)	对元素在文档坐标中的位置进行设定 (*1)
offset(func(index, pos))	将函数 func 的返回值作为元素在文档坐标中的位置进行设定。 函数将接收元素的下标与当前的位置，并返回将要设置的位置
scrollTop()	获取元素的纵向滚动位置。单位为 px
scrollTop(val)	将数值设置为元素的纵向滚动位置。单位为 px
scrollLeft()	获取元素的横向滚动位置。单位为 px
scrollLeft(val)	将数值设置为元素的横向滚动位置。单位为 px

(*1) 位置是一个具有 top 属性与 left 属性的对象。各个属性的值都是数值类型（单位为 px）

样式操作方法中的 css() 方法有着多种用途。在向 css() 方法传递了属性名与属性值以执行该方法时，还可以以 "+=10" 或 "-=20" 这样的方式表示所传递的值。例如，如果要让 div 元素的 margin 增加 10px，则可以像下面这样书写。

```
$('div').css('margin', '+=10');
```

12.7.2 动画

jQuery 不仅提供了一些更改样式的简单方法，还准备了不少能够在执行动画的同时更改样式的方法。这些方法常常能够接收以下参数：动画的持续时间（duration）、动画的加速度（easing），以及动画结束时的处理方式（callback）。

■ duration

duration 是动画从开始到结束所需的时间，其单位为毫秒。如果没有指定值，则默认为 400 毫秒。此外，也可以不直接指定数值，而是使用 slow 或 fast 这样的关键字。slow 相当于指定为 600 毫秒，而 fast 则相当于指定为 200 毫秒。

■ easing

easing 是用于设定动画变化的改变量的关键字。标准的做法是使用 swing 与 linear 这两个关键字来设定。如果没有指定值，则默认为 swing。

swing 将会以“最初缓慢、中途快速、最后缓慢”的方式对值改变。其具体实现是通过数学中的 cos 函数完成的。

而 linear 则会与所经过的时间成比例地进行线性变化。假如每秒移动 100px，那么，经过了 0.1 秒时将会移动 10px，经过了 0.5 秒时移动 50px，而经过了 0.99 秒时移动了 99px。

如果使用了 jQuery UI，则还能够使用更多的关键字。

■ callback

callback 指定的是在动画结束时将执行的回调函数。该回调函数不接收任何参数。而在回调函数内的 this 是正在执行动画的 DOM 元素。

表 12.5 总结了可以用于动画的方法。而如果使用了 jQuery UI，则能够简单地实现丰富多彩的动画效果。

表 12.5 动画

方法	说明
hide([duration,] [easing,] [callback])	隐藏元素
show([duration,] [easing,] [callback])	显示元素
toggle([duration,] [easing,] [callback])	切换元素的隐藏・显示状态
fadeIn([duration,] [easing,] [callback])	对元素执行淡入效果
fadeOut([duration,] [easing,] [callback])	对元素执行淡出效果
fadeToggle([duration,] [easing,] [callback])	如果元素被隐藏则淡入，否则淡出

（续）

方法	说明
fadeTo(duration, opacity, [easing,] [callback])	将元素的不透明度渐变至 opacity
slideDown([duration,] [easing,] [callback])	向下滑动元素
slideUp([duration,] [easing,] [callback])	向上滑动元素
slideToggle([duration,] [easing,] [callback])	如果元素被隐藏则向下滑动，否则向上滑动
animate(props, [duration,] [easing,] [callback])	将元素的 CSS 变为 props 的状态
stop([clearQueue,] [jumpToEnd])	停止动画
queue([queueName])	取得队列
queue([queueName], newQueue)	向队列中添加处理
dequeue([queueName])	去除队列中的第一个元素并执行该元素
delay(duration, [queueName])	以 duration 为时间暂停队列的处理
clearQueue([queueName])	清空队列

jQuery.fx.interval 与 jQuery.fx.off 是与在 jQuery 中执行的动画全体相关的两个属性。

■ jQuery.fx.interval

jQuery.fx.interval 指定了动画的帧间隔，其默认值为 13 毫秒。如果增大该值，动画将变得卡顿。如果减小该值，动画则会变得流畅，不过这当然也取决于 CPU 等客户端 PC 的处理性能。无法对每个动画分别设定帧间隔。

■ jQuery.fx.off

在将 jQuery.fx.off 指定为 true 之后，所有的动画效果都将被禁用。该属性可用于移动设备等客户端性能较低的场合，以期提高在特定环境下的可用性。如果 jQuery.fx.off 为 true，在所有的动画中指定的执行时间都将被忽略。

12.8 通过 jQuery 进行 AJAX 操作

12.8.1 AJAX() 函数

在 jQuery 中，我们可以通过 jQuery.ajax() 函数来实现 AJAX 处理。可以对 ajax() 函数指定 2 个参数。第 1 个参数指定的是目标 URL。第 2 个参数则是一个被指定了相关参数、所使用的 HTTP 类型或回调函数等信息的对象（代码清单 12.8）。此外，也可以省略第 1 个参数，而将 URL 指定为第 2 个参数中的对象的属性。

代码清单 12.8　通过 jQuery 执行 AJAX 操作

```
$.ajax('/foo', {
    type: 'GET',
    success: function (data, status, xhr) {
        // 成功时将执行的处理
    },
    error: function (xhr, status, errorThrown) {
        // 失败时将执行的处理
    }
});
```

表 12.6 总结了传递给第 2 个参数的对象可以指定的一些主要属性。

表 12.6　在 ajax() 函数中可以指定的属性

属性名	说明
url	请求发送目标的 URL
type	所使用的 HTTP 类型

（续）

属性名	说明
timeout	超时时间。单位为毫秒
async	是否执行异步通信
crossDomain	是否执行跨源通信
isLocal	在访问文件系统等本地环境时值为 true
data	所发送的数据对象或字符串
processData	是否不将 data 转换为查询字符串就发送
traditional	该属性用于对将 data 转换为查询字符串时所用的序列化方式进行指定。如果被指定为 true，则通过老式的方式，在转换时不对嵌套的对象进行序列化
headers	请求头部
ifModified	如果该值被指定为 true，则只有在数据被更改的时候请求才算发送成功
cache	是否使用浏览器缓存
dataType	通过字符串指定响应数据的类型。可指定为 xml、html、script、json 及 text 中的一种。回调函数将会把数据转换为这里所指定类型后传递
accepts	用于对 Accept 头部信息进行指定的值的映射。其键为 dataType，值为 Accept 头部信息所指定的值
mimeType	强制覆写于 Content-Type 头部信息中的值
contents	在通过 Content-Type 头部信息来判断响应数据的类型时所用的正则表达式的映射（其键为 dataType，其值为正则表达式）
converters	用于分析响应数据的函数的映射。其键为转换前与转换后的数据类型以空格相连而成的字符串（如果是由 text 转换为了 html 则是 "text html"），其值为函数
context	回调函数内的 this 所引用的对象
beforeSend(xhr, settings)	在发送前执行的回调函数。如果该函数返回了 false，则将取消请求的发送
success(data, status, xhr)	在通信成功时执行的回调函数
error(xhr, status)	在通信失败时执行的回调函数
complete(xhr, status)	在通信完成时执行的回调函数。在成功或失败时也将执行
dataFilter(data, type)	用于对响应数据过滤的回调函数。该函数将在 success() 之前被执行，其结果作为 data 参数传递给 success()
statusCode	用于指定每一个状态码的回调函数的映射。其键为状态码，其值为函数
jsonp	在发送 JSONP 请求所使用的用于指定回调函数名的参数名。如果没有指定，则默认参数名为 callback
jsonpCallback	JSONP 请求的回调函数名。如果没有指定，则会自动设定
scriptCharset	指定读取 script 时所用的字符集。仅在 dataType 为 jsonp 或 script 时有效
global	是否触发与 AJAX 相关的全局事件
xhr	用于创建 XMLHttpRequest 对象的工厂函数
xhrFields	XMLHttpRequest 对象所设定的属性的映射
username	在需要认证的访问中所用的用户名
password	在需要认证的访问中所用的密码

12.8.2 AJAX() 的包装函数

ajax() 可以制定各种各样的选项，不过对于一些常用的设定与处理，jQuery 备有专用的包装函数。表 12.7 对包装函数进行了总结。

表 12.7 ajax() 函数的包装函数

函数	说明
get(url, [data,] [success(data, status, xhr)] [dataType])	通过 GET 方式进行通信。 可以对发送的数据、成功时的回调函数，及响应的数据类型进行指定
post(url, [data,] [success(data, status, xhr)] [dataType])	通过 POST 方式进行通信。 参数与 get() 函数相同
getJSON(url, [data,] [success(data, status, xhr)])	通过 GET 方式获取 JSON 数据。 可以对发送的数据、通信成功时的回调函数进行指定
getScript(url, [success(data, status, xhr)])	通过 GET 方式获取 JavaScript 并执行。 可以对通信成功时的回调函数进行指定

此外，可以通过 ajaxSetup() 函数来更改 ajax() 函数中所能设定的选项的默认值。于是可以先使用 ajaxSetup() 设定各项细节选项，然后在实际的通信中使用包装函数，来实现通信处理。

12.8.3　全局事件

在使用 ajax() 函数时将会触发多个事件。由于这些事件对任何元素都有效，因此被称为全局事件。如果不希望触发全局事件，则需要通过 ajax() 函数的选项将 global 属性设置为 false。

表 12.8 总结了用于向全局事件注册事件侦听器的方法。

表 12.8　全局事件的事件侦听器注册方法

方法名	侦听器的执行时机
ajaxStart(func())	AJAX 通信开始时。 即使有多个 AJAX 通信同时执行，也仅会被执行 1 次
ajaxSend(func(event, xhr, options))	请求发送前
ajaxSuccess(func(event, xhr, options))	通信成功时
ajaxError(func(event, xhr, options, error))	通信失败时
ajaxComplete(func(event, xhr, options))	通信结束时（无论通信成功还是失败）
ajaxStop(func())	所有的 AJAX 通信结束时

12.9　Deferred

Deferred 是一种将异步处理串联书写并执行的机制。

在进行同步处理时，各种处理将会按照代码的书写顺序执行。在一个处理结束之前，下一个处理不会执行。与之相对的，再进行异步处理时，即使书写时是具有一定的顺序的，也会同时进行多个处理，因此很难指定处理的顺序，让异步处理 B 在异步处理 A 完成之后再开始进行。虽然也可以通过将之后要进行的处理指定为回调函数以实现对处理顺序的设定，但这样一来回调函数将会产生多重嵌套，书写会变得很复杂。而且，如果希望在处理 A 与处理 B 结束之后再处理 C，复杂程度就会更高了。而 Deferred 则是一种用于简单地描述这类处理的机制。

12.9.1　Deferred 的基本概念

首先，代码清单 12.9 是一个使用了 Deferred 的代码示例。这里的代码实现了在异步处理 foo 结束之后再执行另一个处理 bar。

代码清单 12.9　使用了 Deferred 的代码示例

```
function foo() {
    var d = $.Deferred();      // 创建 Deferred 对象
    // 异步处理
    setTimeout(function() {
        d.resolve();           // 向 Deferred 报告处理已经结束
    }, 1000);
    return d.promise();        // 返回 Promise 对象
}
function bar() {
    alert('bar');
};

// 在 foo 结束之后开始执行 bar
foo().then(bar);
```

在代码清单 12.9 中出现了 Deferred 对象与 Promise 对象。

■ Deferred 对象

Deferred 对象是一种具有 unresolved、resolved、rejected 中的某一种状态的对象，其初始状态为

unresolved。而只要状态为 unresolved，就不会执行之后的处理。只有当状态变为 resolved 或 rejected 之后，才会开始执行之后的处理。

Deferred 的内部机制为，先注册回调函数，并在 Deferred 对象的状态发生变化时执行该函数。不过在使用 Deferred 的代码时，不必对其内部实现方式在意太多，只要知道它将在某一处理结束之后执行下一个处理即可。Deferred 就是一种所谓的用于提高可读性的机制，可以让异步处理的连锁处理以一种简单易懂的源代码来书写，之后本书将从使用 Deferred 的角度来说明。因此，虽然后续处理实际上只是一种回调函数，但在这里将会把它称为后续函数。

■ Promise 对象

将 Deferred 对象中的一部分方法删除之后，就得到了 Promise 对象。如果直接返回一个 Deferred 对象，收到对象的函数可能会随意修改其状态。为了防止这种情况而定义了 Promise 对象。对状态的管理还是应该由最初创建了该 Deferred 对象的所有者来执行。

12.9.2 状态迁移

Deferred 对象最初的状态是 unresolved。可以通过 resolve() 方法将状态迁移至 resolved。同样地，可以通过 reject() 方法将状态迁移至 rejected。两者都能够接收任意的参数。这里所指定的参数将会在之后直接传递给后续函数。

此外还有 resolveWith() 与 rejectWith() 方法。它们的第 1 个参数可以接收后续函数中的 this 所引用的对象。第 2 个参数则用于接收将直接传递给后续函数的参数的数组。

Deferred 对象在进行了一次状态迁移之后就不能再迁移至另一个状态了。状态的迁移只能发生一次。另外，可以通过 isResolved() 及 isRejected() 方法对状态进行查询。

代码清单 12.10 是一个使用了状态迁移方法的示例。

代码清单 12.10 状态迁移方法

```
var foo = function() {
    var d = $.Deferred() {
        // 将状态设定为 resolved
        // 指定 3 个参数
        d.resolve('This is', 'resolved', 'deferred.');
    }, 500);
    return d.promise();
};

var bar = function() {
    var d = $.Deferred();
    setTimeout(function() {
        // 将状态设定为 rejected
        // 参数指定为后续函数中的 this 与两个参数
        d.rejectWith({
            message: 'This is %s %s'
        }, ['rejected', 'deferred.']);
    }, 500);
    return d.promise();
};

// done() 方法用于指定在 resolved 时执行的后续函数
foo().done(function(arg1, arg2, arg3) {
    console.log('%s %s %s', arg1, arg2, arg3);
});

// fail() 方法用于指定在 rejected 时执行的后续函数
bar().fail(function(arg1, arg2) {
    console.log(this.message, arg1, arg2);
});
```

```
/*
在 console 中所显示的结果

This is resolved deferred.
This is resolved deferred.
*/
```

12.9.3　后续函数

then()、done()、fail()、always() 与 pipe() 是用于指定后续函数的方法。

■ then()、done()、fail()、always()

done() 用于指定状态变为 resolved 时所执行的处理。fail() 用于指定状态变为 rejected 时的处理。then() 所指定的处理则在两种情况下都会执行。then() 的第 1 个参数是 resolved 时的处理，而第 2 个参数则是 rejected 时的处理。always() 所指定的处理在 resolved 与 rejected 这两种情况下都会执行。

这些方法可以多次执行。这时，将会以方法执行的顺序来执行后续函数。此外，对于状态已经变为了 resolved 的 Deferred 对象，在执行 done() 方法时将会立即执行相应的后续函数。

代码清单 12.11 是一个使用示例。

代码清单 12.11　指定后续函数的方法

```
var foo = function() {
    var d = $.Deferred();
    setTimeout(function() {
        // 将状态设定为 rejected
        d.reject('foo');        // 将 'foo' 设定为将会传递至后续函数
    }, 500);
    return d.promise();
};

foo()
.done(function(arg) {
    console.log(arg + ' success 1');
})
.fail(function(arg) {
    console.log(arg + ' success 2');
}, function(arg) {
    console.log(arg + ' failure 2');
})
.always(function(arg) {
    console.log(arg + ' complete 1');
})
.always(function(arg) {
    console.log(arg + ' complete 2');
});

/*
在 console 中所显示的结果

foo failure 1
foo failure 2
foo complete 1
foo complete 2
*/
```

■ pipe()

pipe() 与 then() 相同，都会接收状态变为 resolved 时的处理与状态变为 rejected 时的处理这两个参数。不过，pipe() 与其他方法的使用方式有些不同。

pipe() 具有两种功能。其一是更改参数的值。对于 done() 方法指定的后续函数，将直接使用 Deferred 对象的 resolve() 方法传来的参数。而通过 pipe() 则能够更改这些参数的值（代码清单 12.12）。

代码清单 12.12　通过 pipe() 来改变参数的值

```
var d = $.Deferred();
var filtered = d.pipe(function(arg) {
    return arg * 100;
});
d.resolve(100);

filtered.done(function(arg) {
    alert(arg);  // => 10000
});
```

另一个功能是 Deferred 对象链。

对于 done() 与 fail()，说到底是根据起始的 Deferred 对象的状态来执行的。试考虑代码清单 12.13 中的处理。

代码清单 12.13　通过 done() 执行后续函数

```
var foo = function() {
    // 创建 Deferred 对象
    var d = $.Deferred();
    // 异步处理
    setTimeout(function() {
        alert('foo');
        d.resolve();
    }, 500);
    return d.promise();
};

var bar = function() {
    // 创建 Deferred 对象
    var d = $.Deferred();
    // 异步处理
    setTimeout(function() {
        alert('bar');
        d.resolve();
    }, 1000);
    return d.promise();
};
var baz = function() {
    // 同步处理
    alert('baz');
};

// 将会以 foo、baz、bar 的顺序显示提示
foo().done(bar).done(baz);
```

在代码清单 12.13 的处理中，有人可能会认为提示的显示顺序是 foo、bar、baz，但实际上却是以 foo、baz、bar 的顺序显示的。发生这种问题的原因是 done() 中设定的后续函数最终还是根据 foo 中创建的 Deferred 对象的状态来执行的。因此，在 foo 显示提示时，bar() 与 baz() 将以可能执行的顺序执行。这时，bar 将会在 1 秒后显示提示，而与之相对地，baz 将立即显示提示，于是首先显示的将是 baz。这里很重要的一点是，baz() 并不是在 bar 的提示显示时才开始执行，它实际上是在 foo 显示了提示之后就能执行。也就是说，即使在 done() 所指定的后续函数内创建了新的 Deferred 对象，它也不会影响到之后的后续处理。在这一点上，fail()、then()、always() 也是一样的。

而与之相对地，使用 pipe() 就能够实现和所预期的相同的执行结果（代码清单 12.14）。

代码清单 12.14　通过 pipe() 执行后续函数

```
// 将会以 foo、bar、baz 的顺序显示提示
foo().pipe(bar).done(baz);
```

在 pipe() 之后的处理将会以 pipe() 所指定的后续函数所返回的 Deferred 对象（Promise 对象）为基准。这即是 Deferred 对象链。如果 pipe() 所指定的后续函数返回了 null，则会直接使用前一个状态的 Deferred 对象。

12.9.4　并行处理

可以通过 when() 函数来使后续处理延迟至其他多个异步处理全部完成之后再执行。代码清单 12.15 是一个例子。这里虽然使用了 AJAX 函数，不过其实 AJAX 函数的返回值也是一个 Deferred 对象（Promise 对象）。因此即使不对 AJAX 函数的参数指定回调函数，也能通过 done() 或 fail() 来指定通信结束时的处理。

在代码清单 12.15 的例子中，通过 AJAX 获取了 foo 与 bar 两者，并会在成功时显示“载入成功”的提示，在失败时显示“载入失败”的提示。

代码清单 12.15　通过 when() 函数执行并行处理

```
$.when($.get('/foo'), $.get('/bar'))
.done(function() {
    alert(' 载入成功 ');
})
.fail(function() {
    alert(' 载入失败 ');
});
```

在通过 when() 函数指定了多个 Deferred 对象时，如果有 1 个是 resolved 而其他的是 rejected 的话，将会执行 done() 所指定的后续函数还是会执行 fail() 所指定的后续函数呢？在这种情况下，将会执行的是通过 fail() 指定的后续函数。

通过 done() 指定的后续函数只有在所有的 Deferred 对象都是 resolved 时才会执行。只要有一个是 rejected 也会被认为是整个都是 rejected，而执行 fail() 所执行的后续处理。

12.10　jQuery 插件

在 jQuery 中有各种各样的插件。插件的开发很容易，而且还能够简单地使用他人开发的插件，这些也推动了 jQuery 的人气。

jQuery 中的插件数量确实很多。通常所能想到的用于实现 UI 或某些功能的插件基本上都已存在。因此不需要自己开发插件，在大多数情况下都只要搜索一下即可。可以在下面的 URL 搜索并获取 jQuery 插件。在执笔本书时，已经发布了约 5700 个插件。从这一数字也能看出 jQuery 受欢迎的程度。

http://plugins.jquery.com/

12.10.1　使用 jQuery 插件

只要在载入了 jQuery 库之后再载入插件的 JavaScript 库就能够在 jQuery 中使用插件了。本节将以 Jason Frame 开发的插件 tipsy（http://onehackoranother.com/projects/jquery/tipsy/）为例，对插件的使用方式进行说明。

tipsy 是一个可以在气泡式窗口中显示工具提示信息的简单的插件。元素的 title 属性所设定的值将会

作为提示信息的内容被显示出来。而根据具体的设定，还能够显示其他属性的值。此外，还能够对提示信息的显示方向、以及显示信息出现与消失时的淡入淡出等效果进行指定。

代码清单 12.16 是一个使用了 tipsy 的代码示例。

代码清单 12.16 jQuery 插件的使用示例

```
<!DOCTYPE HTML>
<html lang="zh-CH">
<head>
    <meta charset="UTF-8">
    <link rel="stylesheet" type="text/css" href="tipsy/stylesheets/tipsy.css" />
    <!-- 先要载入 jQuery -->
    <script src="jquery.js"></script>
    <!-- 在载入了 jQuery 之后再载入 tipsy 插件的文件 -->
    <script src="tipsy/javascripts/jquery.tipsy.js"></script>
    <script>
        $(function() {
            // 由于使用了 tipsy 插件而可以在 jQuery 对象中使用插件定义的 tipsy 方法
            // tipsy 具体的使用方法可以参见其官方站点说明或其他相关资料①
```

tipsy 的执行效果如图 12.1 所示。

图 12.1 jQuery 插件的使用示例



12.10.2 创建 jQuery 插件

可以通过扩展 jQuery.fn 来创建 jQuery 插件。jQuery 插件的创建方法如代码清单 12.17 所示。

代码清单 12.17 jQuery 插件的模板

```
(function($) {
    $.fn.myplugin = function(method) {
        var methods = {
            init: function(options) {
                this.myplugin.settings = $.extend({}, this.myplugin.defaults, options)
                return this.each(function() {
                    // 初始化处理
                });
            },
            someMethod: function() {
                // 插件专有的函数
            }
        }
        if (methods[method]) {
            return methods[method].apply(this, Array.prototype.slice.call(arguments, 1));
        } else if {typeof method === 'object' || !method} {
```

① 这里有个让人费解的地方。代码清单 12.16 是不完整的，像是中途中断了，这与本书一直以来的风格不同。而且由于代码的缺失，也无法从代码中得出下图的结果。从内容上来说，也并没能清晰地告诉读者究竟应该如何使用该插件。因而译者擅自在最后增加了那一句注释以帮助说明这一情况。——译者注

```
            return methods.init.apply(this, arguments);
        } else {
            $.error('Method "' + method + '" does not exsit in myplugin plugin!');
        }
    }

    $.fn.myplugin.defaults = {
        foo: 'bar'
    }
    $.fn.myplugin.settings = {}
}(jQuery));
```

可以像下面这样来执行该插件。

```
$('selector')
    .myplugin({ foo: 'baz' })       // 设定插件
    .myplugin('someMethod');        // 执行 someMethod
```

12.11 与其他库共同使用

12.11.1 $ 对象的冲突

jQuery 将会在全局作用域中创建 jQuery 对象与 $ 对象。除此之外不会创建其他多余的对象，也不会对 prototype 造成污染，是一个便于使用的库。不过，$ 这一名称在很多的 JavaScript 库中都会用到。

最初，Prototype.js 将 $ 当成 Document.getElementById() 的别名，并作为一种选择器函数使用。因此，在 JavaScript 中将 $ 作为选择器函数来使用逐渐成为了一种约定俗成的做法。常常能看到下面这样的 $ 函数的习惯用法。

```
function $(element) {
    return document.getElementById(element);
}
```

在 Prototype.js 之后的很多 JavaScript 库中，也定义了很多将 $ 函数作为选择器使用的方法。因此，如果同时使用了多个库的话，$ 这一名称就很有可能会发生冲突。例如，如果同时使用 jQuery 与 Prototype.js，$ 函数所引用的将会是最近被定义的那个对象（代码清单 12.18）。

代码清单 12.18　同时使用 jQuery 与 Prototype.js

```
<!-- jQuery 在前，Prototype.js 在后 -->
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.6.2/jquery.js"></script>
<script src="https://ajax.googleapis.com/ajax/libs/prototype/1.7.0.0/prototype.js"></script>
<script>
    alert($);    // 将显示 Prototype 的 $ 函数
</script>

<!-- Prototype.js 在前，jQuery 在后 -->
<script src="https://ajax.googleapis.com/ajax/libs/prototype/1.7.0.0/prototype.js"></script>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.6.2/jquery.js"></script>
<script>
    alert($);    // 将显示 jQuery 的 $ 函数
</script>
```

12.11.2 避免 $ 对象的冲突

我们可以在 jQuery 中通过 noConflict() 方法来避免名称冲突。在使用了 noConflict() 方法之后，jQuery 所定义的 window.$ 对象就会被删除，而能够使用原先被定义的那个 $ 对象（代码清单 12.19）。顺

便说明一下，在 Prototype.js 中没有用于避免名称冲突的方法，所以只能在 jQuery 中避免这一问题。

代码清单 12.19 避免名称冲突

```
jQuery.noConflict();                  // 删除 window.$
var $j = jQuery.noConflict(true);    // 同时删除 window.$ 与 window.jQuery 两者。返回值是一个 jQuery 对象
```

通常并不需要使用 noConflict(true)。如果自己对 window.jQuery 对象进行了定义，或者有多个不同版本的 jQuery 共存，可能会要用到这一方法。不过尽可能避免出现这种情况为好。一方面没必要特意创建一个与 jQuery 的名称冲突的对象，另一方面应该也能通过别的方法来避免出现多个版本的 jQuery 共存的情况。此外，如果 window.jQuery 处于未被定义的状态，就无法添加 jQuery 插件。因此，如果非要使用 noConflict() 的话，请至多使用其不含参数的版本。

12.12 库的使用方法

为了使用一个库，必须要载入该库的 JavaScript 文件。由于库通常都会使用允许再次发布的许可，因此就算将库文件置于自己的服务器上让用户对其进行访问也自然是没什么问题。不过，也可不必特意将库放在自己的服务器上，而可以通过 Google Libraries API 来读取 JavaScript 库。

http://code.google.com/apis/libraries/

Google Libraries API 是一个 JavaScript 的内容分发网络（CDN）。由于这是 Google 提供的服务，因此几乎不会有任何网络或服务器故障，至少会比自己准备的服务器或网络可靠性更高。

表 12.9 总结了执笔本书时 Google Libraries API 所提供的 JavaScript 库。其中几乎包含大部分著名的 JavaScript 库。

表 12.9 在 Google Libraries API 中提供的 JavaScript 库

库	说明
Chrome Frame	用于对是否安装了 Google Chrome Frame（能够使 Internet Explorer 的执行引擎实现与 Google Chrome 相同效果的插件）进行确认的库
Dojo	全栈式框架的 JavaScript 库
Ext Core	Ext JS 的核心部分。Ext JS 是一个以丰富的 UI 组件为特点的库
jQuery	其特点为通过链式语法实现以简单的代码对 DOM 进行操作
jQuery UI	在 jQuery 的基础上添加 UI 效果和 UI 组件的库
MooTools	针对中高级开发人员设计的轻量级库
Prototype.js	最早流行的 JavaScript 库
script.aculo.us	在 Ptototype 的基础上添加了动画等 UI 效果的库
SWFObject	用于将 Adobe Flash 嵌入 HTML 的库
YUI Library	Yahoo! 提供的全栈式框架的库
WebFont Loader	用于使用 Web 字体的库

不仅 Google，Microsoft 也提供了同样的 CDN 服务，不过在其 CDN 中只提供了 jQuery 这一个库。因此，如果要使用 jQuery 以外的库的话，应该选择 Google Libraries API。

此外还有一个名为 cdnjs.com 的 CDN 服务，提供了在表 12.9 中没有被记载的库。该服务用于提供那些 Google 的 CDN 中没有提供的不那么著名的库。如果希望通过 CDN 来使用最近新出现的库的话，可以试着在这里寻找一下。

http://www.cdnjs.com



第 4 部分

HTML5

本部分介绍 HTML5 这一由 W3C 制定的标准及其相关技术。其中，将会重点解说在 HTML5 繁杂的标准中，那些将会对今后的 JavaScript 开发产生重大影响的 API。

第 13 章 HTML5 概要

我们首先概述 HTML5。本章的目的是理解 HTML5 的制定背景以及现状，以期在整体上把握其所提供的新功能。

13.1 HTML5 的历史

HTML5 的发展历程

我们先简单回顾一下 HTML5 至今的发展历程。在第 1 章中已经对 JavaScript 的历史做了介绍。与前几年相比，浏览器中所采用的 JavaScript 实现性能大为提升，于是浏览器也成为了一个应用程序平台，而受到了人们的瞩目。

随着 Web 应用程序的范围越来越广，JavaScript 的功能局限性也渐渐引起了人们的注意。仅靠 JavaScript 无法实现数据的永久保存、套接字通信、音乐与视频的播放等在桌面应用程序中常见的功能，于是在一些特定领域，Web 应用程序的发展要慢于桌面应用程序。

■ 浏览器插件的普及

在此之前，为了弥补 JavaScript 所缺少的功能，采用的是 Flash 或 Silverlight 等插件。也就是说，尽管有这样的需求，但这些功能并没有被作为浏览器的标准功能被提供。

即使如此，如果有浏览器开发商率先实现了某一个新的功能，但没能在所有的主流浏览器中得到支持的话，这一服务还是不能得到进一步利用。因此，从功能扩充的角度来说，在很长一段时间里，仍然需要在浏览器中安装 Flash 及 Silverlight 等插件。

■ 统一的浏览器标准的制订

HTML5 就是在那些不满于现状的浏览器开发商的共同合作下诞生的。为了更有效率地实现功能的扩展，必须要有统一的标准。

但是，现在广泛普及的 HTML 的标准制定止于 1999 年 W3C[1] 所推荐的 HTML4.01 版本。虽然 W3C 对 XHTML 的普及付出了很多的努力，但最终 XHTML 并没能像预期的那样得到普及。因此，对 W3C 的方式怀有不满的 Apple、Mozilla、Opera 成立了名为 WHATWG[2] 的社区。WHATWG 的目的是通过与当前 Web 发展状况相适应的方法，制订与 Web 相关的技术标准，并将其反馈给 W3C。

■ HTML5 标准制订的起步

在历经这一发展阶段后，W3C 终于与 WHATWG 开始了合作，制订 HTML5 的标准，并在 2008 年 1 月 22 日发表了第一份 HTML5 草案。在执笔本书时，最终版本的草案（2011 年 5 月 25 日）已发表，今后 W3C 将按照下面的进程逐步使 HTML5 成为推荐标准，并以 2014 年完成这一推荐为目标。

- 草案（Working Draft）

① W3C（World Wide Web consortium）是一个推进 WWW 中所使用的各种技术的标准化的团体（http://www.w3.org/）

② http://www.whatwg.org/

- 最终草案（Last Call Working Draft）
- 推荐候选（Candidate Recommendation）
- 推荐方案（Proposed Recommendation）
- 推荐（Recommendation）

13.2　HTML5 的现状

13.2.1 浏览器的支持情况

■ PC

当前，针对 PC 的浏览器市场正处于一种 Internet Explorer（IE）、Firefox、Chrome、Safari 及 Opera 等浏览器在相互争夺份额的情况。如果希望在这些针对 PC 的浏览器中提供使用了 HTML5 的服务的话，需要注意一个问题，即 IE（6、7、8）几乎不支持任何的 HTML5 的相关功能，并且它们的市场占有率相当高，让人无法忽略。

Microsoft 声称将从 IE9 开始加强对 HTML5 的支持，并且也确实在 IE9 中实现了很多的 HTML5 的相关功能。在执笔本书时，IE10 的预览版也已公开。不过就此前的趋势来看，旧版本的 IE 的市场份额还需要很长的时间才会降低到忽略不计的程度吧。

因此，如果要通过 HTML5 来提供针对 PC 的服务的话，则需要区分不同的浏览器。但是，在这种情况下，检查浏览器的种类或版本等并替换处理方式的做法不易管理，所以通常会在执行 JavaScript 时判断所需功能是否被支持以选择相应的处理方式。目前，可以通过 Modernizr[①] 等工具能够实现这一功能。

■ 智能手机

如果将目标限定于智能手机，情况将与 PC 有很大的不同。之所以这样说，是因为市场份额正在迅速增加的 iOS 与 Android 这两个针对智能手机的 OS 都默认使用了基于 WebKit[②] 的浏览器。特别是在日本，iOS 与 Android 占据了非常高的市场份额[③]，所以只要对 WebKit 提供了支持，就能够支持几乎所有的智能手机了。

因此，在开发针对智能手机的 Web 应用程序时，几乎不需要考虑跨浏览器支持的问题。甚至可以说，智能手机是现在最容易进行 HTML5 开发的领域。

■ 电视机

并不是只有 PC 与智能手机才会安装浏览器。现在日本厂商制造的很多电视机中都装有基于 NetFront 这一针对嵌入式设备设计的浏览器的定制版浏览器。目前这些浏览器几乎还不支持 HTML5 相关的功能。

最近索尼终于在其电视机中预置了 Opera 浏览器，以此为开端，使用 HTML5 的功能强大的 Web 应用程序开发环境也日趋成熟。加上 Apple TV 与 Google TV 等产品的开发，电视机浏览器这一领域今后的动向也受到了人们的瞩目[④]。

---

① http://modernizr.com/

② WebKit（http://www.webkit.org/）是以苹果公司为中心所开发的开源 HTML 渲染引擎。被 Safari 或 Google Chrome 等很多的浏览器所使用。

③ 事实上，在包括中国在内的全球范围内都是如此。——译者注

④ 尽管 NetFront 浏览器被用于包括曾经流行过的 Plam OS、Windows CE 以及一些日本厂商所开发的游戏主机在内的各种设备中，但现在它在日本以外的市场中并没有被广泛使用。现在最新的 4.2 版本的 NetFront 浏览器已经对 HTML5 提供了一定程度的支持，而一些基于 Android 系统的 Google TV 设备也采用了 WebKit 核心的浏览器。此外，目前国内的智能电视采用的也主要是 WebKit 核心的浏览器。——译者注

13.2.2 Web 应用程序与原生应用程序

HTML5 一词似乎已经成了一句流行语，在不同场合下这个词所指的范围也各有不同。有时 HTML5 仅被用来表示严格包含于 W3C 的 HTML5 标准中的内容，不过在本书中，HTML5 代表了更为广泛的内容。最近，为了使原生应用程序可以做到的功能在浏览器中也能够被实现，出现了大量的扩展功能。在本书中，将把这些浏览器的扩展功能统称为 HTML5。

从 Gmail 开始，已经出现了很多以 Web 应用程序的形式提供的实用程序。与过去相比，使用原生应用程序的机会少了很多。而最近的浏览器甚至实现了 3D 图形的渲染功能，这一势头让人不禁觉得总有一天所有的程序都会改以 Web 应用程序的形式存在了吧。

不过，无论浏览器的扩展功能如何发展，只要浏览器还是一种原生应用程序，它在能够实现的功能方面就不会有什么优势。即使如此，支持将浏览器作为一种应用程序平台也是有其理由的，其中之一就是开发的成本问题。

例如，通常情况下在针对智能手机的原生应用开发中都需要使用 OS 开发商所发布的 SDK。大致说来，就是对于 iOS 终端主要使用 Objective-C、对于 Android 设备主要使用 Java、对于 Windows Phone 7[①] 设备则会使用 C# 或 Visual Basic 以进行开发。在程序设计语言与 SDK 的学习、应用程序的实现、今后的后续修改等方面，所花费的成本将会与需要支持的平台数量成比例上升。

而 Web 应用程序正是解决这一领域问题的最佳方案。通过 HTML/CSS/JavaScript 开发的 Web 应用程序能够通过浏览器运行于所有的智能手机上。在目前的智能手机应用开发中，Web 应用程序几乎可说是实现跨平台支持的唯一手段。

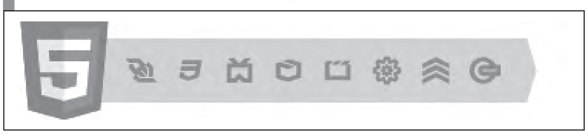
13.3 HTML5 的概要

W3C 为了使 HTML5 得到广泛普及而为它提供了各种图标[②]。使用了 HTML5 及其相关技术的页面或应用程序将显示其徽标以对 HTML5 的使用进行宣传。通过在徽标中列出具体的图标还可以明确地标识出其中具体使用了哪些技术（图 13.1）。

提供了这些图标的站点将图标与相对应的 HTML5 及其相关技术分成了 8 大类，以使人们能对 HTML5 所提供的功能有一个整体的把握。本节将对此作简单介绍。表 13.1 总结了各个图标所表示的技术领域。表 13.2 总结了一些与 HTML5 有关的重要的 JavaScript API 及其简单说明。

与 HTML5 相关的功能不仅仅是 JavaScript，还包括了 HTML、DOM、CSS 等领域。不过本书无法包含以上所有内容，本书将只选择与 HTML5 相关的 JavaScript API，尤其是其中被认为会对今后的 JavaScript 开发产生巨大影响的部分进行说明。

图 13.1 HTML5 徽标（全套）



① 及 Windows Phone 8。——译者注

② http://www.w3.org/html/logo/

表 13.1　HTML5 及其相关技术的分类

类别	说明
CONECTIVITY	WebSocket、Server-Sent Event 等
CSS3	CSS3、Web Fonts 等
DEVICE ACCESS	位置信息、加速度传感器等
3D GRAPHICS & EFFECTS	SVG、Canvas、WebGL 等
MULTIMEDIA	Audio 标签、Video 标签等
PERFORMANCE & INTEGRATION	Web Workers、XHR2 等
SEMANTICS	microdata、microformats、轮廓元素的添加等
OFFLINE & STORAGE	ApplicationCache、localStorage、IndexedDB、File API 等

表 13.2　HTML5 相关 API 一览

API 名称	功能说明
Video/Audio	视频或音频播放
Canvas	2D 图形绘制
WebGL	3D 图形绘制
Web Messaging	窗口间的数据收发
WebSocket	与服务器双向通信
Server-Sent Events	来自服务器的数据推送
Web Workers	后台处理
Web Storage	简单的键值存储
Indexed Database	功能强大的键值存储
Web SQL Database	功能强大的关系数据库
File API	对文件系统的访问
Drag and Drop	拖放操作
Geolocation API	获取当前位置信息
Application Cache	对缓存文件的控制
History API	对浏览器历史的操作

专栏

浏览器开发商的 HTML5 信息门户站点

与 HTML5 相关的 API 的标准已经由 W3C 公布，本书也在对各 API 介绍时附上了 W3C 的相关页面链接。一般来说，参考 W3C 的标准就不会有错了，不过在实际开发中，有很多功能即使读了标准也很难理解其概念。

在这种情况下，我推荐大家参考一下各个浏览器开发商所提供的 HTML5 相关信息的综合门户站点。这些站点上会公开一些实际能够执行的演示程序，与 W3C 提供的标准相比，能够更容易地获取一些实际实现的代码。此外，很多时候浏览器的具体实现会与 W3C 的标准有所不同（例如开发商前缀等），这时就只能从这些站点获取这类浏览器自带的信息了。

首先通过这些开发商提供的门户站点页面掌握功能的概要，之后再去阅读 W3C 的标准，将有助于更快的理解。表 A 总结了主要的浏览器开发商所提供的 HTML5 信息门户站点。

表 A　各浏览器开发商的 HTML5 信息门户站点[①]

开发商	页面名称	URL
Microsoft	IE Test Drive	http://ie.microsoft.com/testdrive/
Mozilla	MDN HTML5	https://developer.mozilla.org/zh-CN/docs/HTML/HTML5
Google	HTML5 Rocks	http://www.html5rocks.com/
Apple	HTML5 Showcase	http://www.apple.com/html5/
Opera	Dev.Opera > open web	http://dev.opera.com/articles/tags/open%20web/

① 原书中有部分站点的 URL 不是最新版，这里已更新为最新的 URL。——译者注

第 14 章 Web 应用程序

本章将讲解 Web 应用程序的 URL 管理以及离线支持，并阐述使用了 HTML5 标准中新加入的 History API 及 ApplicationCache 的更为高级的 Web 应用程序开发方法。

14.1 History API

14.1.1 History API 的定义

History API[1] 是用于在 JavaScript 中对浏览器的 URL 及历史信息进行操作的 API。过去，大部分的 Web 应用程序都是由服务器端负责程序逻辑，客户端则主要负责信息显示。而最近，将复杂的状态变化移至客户端进行管理的 Web 应用程序多了起来。由于在客户端通过 AJAX 方式更新内容时不会改变页面的 URL，因此必须通过 JavaScript 对 URL 进行管理以始终应对页面的状态变化。

然而，如果只是改写页面的 URL，就将会发生页面跳转，且 JavaScript 的状态也会在这时被重置。于是出现了 History API。如果使用 History API，就能够在不发生页面跳转的情况下将 URL 路径替换为任意内容。

14.1.2 哈希片段

■ AJAX 应用程序与哈希片段

原本 URL 的作用是用于唯一识别 Web 上的某一内容。然而对于运用了 AJAX 技术的页面来说，即使不进行页面跳转也能够自由地改写页面内容。此时如果无法对 URL 进行有序的管理，则可能会出现同一个 URL 表示了完全不同的内容的情况。

正如不存在没有地址栏的浏览器，URL 与 Web 应用程序也是不可分割的概念。如果 URL 无法履行其唯一识别 Web 内容的这一原本的功能的话，浏览器的书签功能就将失效，而使用外部内容的链接也将成为一个难题。

为了解决这一问题，现在很多的 AJAX 应用程序都采用了哈希片段（URL 中 # 之后的字符串）这一方式。由于哈希片段使用的就是页面内的链接，因此即使改写了哈希片段也不会发生页面跳转（向服务器发送请求）。利用这一机制以哈希片段来表示页面的状态后，就能够以唯一的 URL 来表示应用程序的特定状态了。

代码清单 14.1 是哈希片段的使用示例。在这个例子中，哈希片段代表了正在浏览的页面的页码信息。需要注意的是，这里的 updateContent 被假定为一个实现了对内容进行更新处理的函数。

代码清单 14.1　哈希片段的使用示例

```
function gotoPage(num) {
```

① http://www.w3.org/TR/html5/history.html

```
        // 更新内容
        updateContent(num);

        // 将当前状态保存至哈希片段
        location.hash = '#!page=' + num;
    }

    // 状态的恢复
    window.onhashchange = function() {
        // 从哈希片段中获取状态
        var num = location.hash.match(/#!page=([0-9]+)/)[1];

        // 更新内容
        updateContent(num);
    };
```

根据规则，在起始处添加了 # 之后，该字符串就将会被识别为哈希片段。不过最近对于上面的示例，使用了”#!(hashbang, shebang)”形式的情况也在增加。

出现这种情况的理由大致有以下两个：其一，这样一来，就能够将哈希片段区别于过去被用于页面内链接的 # 字符串。另一个理由则更为重要，即 Google 提出了一种机制，提议在搜索引擎等的网络蜘蛛中将”#!”之后的内容识别为 AJAX 应用程序的状态以进行处理。

■ 哈希片段与网络蜘蛛

使用哈希片段的 AJAX 应用程序有一个问题，即之前提到的不能很好地支持搜索引擎等的网络蜘蛛。通常网路蜘蛛不会对应用程序中包含的 JavaScript 进行解读，所以无法在获取页面后抓取通过 JavaScript 动态载入的内容。因此，为了让网络蜘蛛能够收集页面的内容，则必须在服务器端识别网络蜘蛛的访问，并返回不包含 JavaScript 的静态内容。

然而 URL 中哈希片段的部分不会和请求一起发送给服务器。也就是说，服务器无法返回与哈希片段所表示的应用程序状态相对应的合适内容，于是，网络蜘蛛也就无法正确获取 URL 原本指示的内容。

为了处理这个问题，Google 提出了一种网络蜘蛛的工作机制。即网络蜘蛛将会把含有 #! 的 URL 识别为 AJAX 应用程序，并将 #! 替换为 ?_escaped_fragnemt_ 这一参数后访问服务器。

```
    // Web 中通常使用的 URL
    http://www.example.com/#!/foo/bar

    // Google 的网络蜘蛛所访问的 URL
    http://www.example.com/?_escaped_fragment_=/foo/bar
```

如果遵守这一规则，只要在服务器端将含有 _escaped_fragment_ 参数的请求识别为来自于网络蜘蛛的访问，就能够根据该参数所示的状态发出相应的静态内容。

一直以来，对于 URL 的处理有着很多争论。随着 AJAX 应用程序的迅速普及，也出现了很多质疑声，他们担心以与原本不同的方式来使用哈希片段是否妥当。而 History API 正是一种可以巧妙地解决这些与 URL 和 AJAX 应用程序相关的问题的 API。

14.1.3　接口

在 History API 中会用到的主要内容有 history 对象与 popstate 对象。history 对象是 window 对象所具有的一个属性，用于对历史记录进行操作。popstate 事件将会在巡览页面历史时被触发，因此可以侦听 popstate 事件，并实现一些用于恢复页面状态的处理。

■ 页面历史的保存

众所周知，我们可以通过 JavaScript 的 location 对象以跳转至特定的页面（代码清单 14.2）。如果不希望发生页面跳转就能够显示特定内容，则可以使用之前介绍的使用了哈希片段以在 URL 中保存页面状态的方法（代码清单 14.3）。

代码清单 14.2 通常的 URL 更新

```
// 发生了指向 /search/foo 的页面跳转
location.href = '/search/foo';
```

代码清单 14.3 使用了哈希片段的 URL 更新

```
// 以 AJAX 的方式获取 /search/foo 的内容并显示
updateContent('/search/foo');
// 将哈希片段更改为 #!/search/foo
location.hash = '#!/search/foo');
```

此时不会发生页面跳转。此外，还可以通过 history 对象的 pushState 方法，以在不使用哈希片段的情况下将 URL 更新为合适的状态（代码清单 14.4）。通过将 URL 指定给 pushState 方法的第 3 个参数，就能够在不向服务器发送请求的情况下更改 URL。而如果指定了完整路径，则应当遵守同源策略（请参见第 16 章的专栏）的限制。

代码清单 14.4 使用 pushState 更新 URL

```
// 以 AJAX 的方式获取 /search/foo 的内容并显示
updateContent('/search/foo');
// 将 URL 更改为 /search/foo
history.pushState(null, 'foo 的搜索结果', '/search/foo');
```

其中第 1 个参数所指定的是用于表示页面状态的详细信息的对象。本节将在之后详细说明该参数。第 2 个参数所指定的是页面的标题。在这里指定的值，可以在诸如显示浏览器的页面浏览历史记录等场合，根据需要从浏览器中引用。

■ 页面历史记录的跳转

可以通过浏览器的后退键以及前进键遍历浏览器所管理的页面历史记录。也可以通过 history 对象的 back 方法与 forward 方法，实现以 JavaScript 来进行页面历史记录的跳转。此外，还可以使用 go 方法以参数所指定的步数在历史记录中向前或向后跳转，不过一般情况下这种方法并不常用。代码清单 14.5 是一个例子。

代码清单 14.5 页面历史记录的跳转

```
// 在历史记录中后退 1 次（等同于浏览器的后退键）
history.back();

// 在历史记录中前进 1 次（等同于浏览器的前进键）
history.forward();

// 在历史记录中后退 2 次
history.go(-2);
```

■ 页面状态的恢复

在遍览通过 pushState 添加的页面历史记录时通常不会发生页面跳转，因此有必要独自对页面状态的恢复处理进行实现。具体来说，由于这时 popstate 事件将被触发，因此可以侦听 popstate 事件，并实现一些用于将页面更新为恰当状态的处理。

页面所显示的内容应当是与 URL 相对应的，因此，在对 popstate 事件进行实现时，基本要点在于应该根据 URL 来执行恰当的内容绘制处理。代码清单 14.6 是一个简化的 popstate 事件的实现示例。

代码清单 14.6 popstate 事件的实现示例

```
// 侦听页面历史记录的跳转
window.onpopstate = function() {
    // 分解 URL 路径
    var pathnames = location.pathname.substring(1).split('/');
```

```
    // 引用顶层的路径名并显示合适的内容
    switch(pathnames[0]) {
        case 'list':
            /* 显示列表页面 */
        case 'search':
            /* 显示搜索页面 */
    }
};
```

■ 恢复更为详细的页面状态

URL 所能保存的信息量是非常少的。通常的 URL 都不会包含诸如“跳转目标的 URL”这样的跨页面信息，或是“层级树的开闭状态”这样的非常细节的页面状态信息。可以借助 pushState 的第 1 个参数，来管理比 URL 所能保存的信息更为详细的页面状态。

例如，试考虑不希望在某一页面中显示用于返回跳转目标的链接的情况。这时跳转目标的 URL 或标题等信息是必需的，但通过 URL 来保存所有这些信息并不是很讨巧的做法。应该像代码清单 14.7 这样，将这些信息传递给 pushState 的第 1 个参数。

代码清单 14.7　对详细状态的管理

```
// 跳转目标的信息
var data = {
    // 跳转目标的标题
    prev_title: document.title,
    // 跳转目标的 URL
    prev_url: location.pathname
};

// 将信息传递给第 1 个参数
history.pushState(data, null, '/foo/bar');
```

可以通过 history.state 属性来引用传递给 pushState 的第 1 个参数的信息。不过这个功能是在最近的更新中出现的，虽然在 Firefox4 以及更新的浏览器中得到了实现，在 Chrome13 中还未被实现，对此请加以注意[①]。

根据原本的标准，只能从 popstate 事件对象中引用 state。然而由于在页面重载时并不会发生 popstate 事件，因此根据这一规则，无法在页面重载时引用 state 并恢复页面的状态。

代码清单 14.8 通过 history.state 引用了传递给 pushState 的第 1 个参数的信息（代码清单 14.7）并以此绘制画面。在这个例子中，始终会引用 URL 与 history.state 来绘制页面内容，也就是说在页面载入时、页面历史记录跳转时、内容更新时等所有的页面跳转中都会使用通用的内容绘制处理方式。

代码清单 14.8　详细状态的恢复

```
// 页面的载入与重载时
window.onload = updateContent;

// 页面历史记录跳转时
window.onpopstate = updateContent;

// 更新内容时
function gotoContent(data, title, pathname) {
    // 添加页面历史记录
    history.pushState(data, title, pathname);
    updateContent();
}

// 引用 URL 及 history.state 并更新内容
```

① 而在 Chrome19 以及更新的版本中也对此提供了实现。——译者注

```
function updateContent() {
    // 引用 URL 并更新内容
    /* 执行一些操作 */

    // 引用 history.state 并更新内容
    if (history.state && history.state.prev_url) {
        // 如果含有信息，则设定后退链接
        backLink.href = history.state.prev_url;
        backLink.textContent = history.state.prev_title || '后退';
        backLink.style.display = '';
    } else {
        // 如果不含信息，则隐藏后退链接
        backLink.style.display = 'none';
    }
}
```

■ 页面历史记录的替换

应该在页面状态被改写时相应地更新 URL。不过如果在保存页面历史记录时分得过细，则需要多次点击后退键才能回到希望回到的状态，反而有可能导致可用性下降。

如果对内容的更新程度没有达到需要添加新的页面历史记录，则可以不添加历史记录而直接覆写正在显示的历史记录信息。可以通过 replaceState 方法来覆盖正在显示的历史记录信息。该方法所需的参数与 pushState 基本相同，只是 replaceState 不会添加新的历史记录而会覆盖当前的历史记录信息。

代码清单 14.9 将在切换复选框的勾选状态时将这一勾选状态写入当前的历史记录信息中。只要恰当地更新历史记录信息，就能够在遍览页面历史记录时正确地恢复复选框的状态。

代码清单 14.9　页面历史记录的替换

```
function toggleCheck(chkbox) {
    // 更改复选框的勾选状态
    chkbox.checked = !chkbox.checked;

    // 复制当前的状态对象
    var data = {};
    for (var prop in (history.state || {})) {
        data[prop] = history.state[prop];
    }
    // 添加勾选状态
    data.chkbox = chkbox.checked;

    // 覆写历史记录信息
    history.replaceState(data, document.title);
}
```

根据标准，history.state 是 read-only 的，所以在这个例子中，先将其赋值给了另一个对象，之后再添加了信息并传递给 replaceState。不过要注意的是，在这个例子中并没有考虑状态对象嵌套的情况。

■ history 对象的属性一览

表 14.1 总结了 history 对象的属性。不过还请大家务必以最新版的标准为准。

表 14.1　history 对象的属性一览

属性名	说明
length	历史记录的总数（包括正在显示的页面）
state	表示当前状态的对象
go(delta)	以 delta 所指定的步数在历史记录中跳转。go(–1) 等同于 back()，go(1) 等同于 forward()
back()	在历史记录中后退 1 次
forward()	在历史记录中前进 1 次
pushState(data, title, [, url])	添加历史记录信息
replaceState(data, title, [, url])	替换当前的历史记录信息

14.2 ApplicationCache

14.2.1 关于缓存管理

在最近的 Web 应用程序开发中，对智能手机的支持已经是一个无法回避的重要事项了。在对智能手机提供支持时，应当考虑的重要的一点是通信线路的不稳定性。移动设备所使用的 3G 线路的通信速度较慢，而且还可能多次发生信号不畅的情况。

通过使用本节将要介绍的缓存清单文件以及 ApplicationCache API，就能够将过去由浏览器进行管理的缓存文件改由应用程序的开发者来控制。利用缓存来减少不必要的文件下载后就能够改善通信速度较慢的问题，而如果使用得当，甚至还能够开发出可以离线使用的 Web 应用程序。

表 14.2 列出了一些支持 ApplicationCache API 的浏览器。IE9 没有提供对该 API 的支持虽然很是可惜，但对于受益于这一功能的智能手机来说，自然是几乎所有的终端都能支持这一 API。

表 14.2 ApplicationCache API 的支持情况

浏览器	版本
Chrome	5.0 及以上
Firefox	3.5 及以上
Safari	4.0 及以上
Opera	10.6 及以上
iOS	2.1 及以上
Android	2.0 及以上

14.2.2 缓存清单文件

■ 缓存清单文件的创建

通过创建被称为缓存清单的文件，就可以设定未被缓存的文件。缓存清单文件实际上就是一个记录了缓存规则的简单的文本文件。不过仅通过这样的说明大家可能还无法理解这一概念，本节之后将通过简单的范例依次对其进行说明。

在代码清单 14.10 的范例中，缓存了 HTML 所引用的所有文件，以期能够进行离线浏览。首先，在 html 标签的 manifest 属性中指定了缓存清单文件的路径。虽然缓存清单文件并没有规定特定的扩展名，不过推荐使用 .appcache。

不过，必须通过 text/cache-manifest 这一 MIME Tpye 来发布缓存清单文件。如果正在使用 Apache，则将在缓存清单文件所在的同一个文件夹内创建一个名为 .htaccess 的文件并记录 AddType 目录信息，以设定支持特定扩展名的 MIME Type（代码清单 14.11）。

代码清单 14.10 cache.html

```
<!DOCTYPE HTML>
<html manifest="sample.appcache">
<head>
    <meta charset="UTF-8">
    <script src="cache.js"></script>
    <link rel="stylesheet" href="cache.css">
</head>
<body>
    <h1>Cache Sample</h1>
    <img src="html5-badge.png">
</body>
```

```
</html>
```

代码清单 14.11 .htaccess 的代码示例

```
AddType text/cache-manifest .appcache
```

在 cache.html（代码清单 14.10）中引用了 cache.js、cache.css、html5-badge.png 这三个文件。可以像代码清单 14.12 这样在缓存清单文件中列举其文件路径，将它们设定为缓存对象。而指定了 manifest 属性的文件将会被自动缓存，所以不需要列出 cache.html。

代码清单 14.12 sample.appcache

```
CACHE MANIFEST
# revision 1

CACHE:
./cache.js
./cache.css
./html5-badge.png
```

必须在缓存清单文件的第一行写上 CACHE MENIFEST。而以 # 开始的行将被识别为注释。写有 CACHE: 的行及其之后的内容是 CACHE 区段，在 CACHE 区段中列举的文件都将被自动缓存。

在准备好了缓存清单文件后就可以试着打开 cache.html 了。在第一次访问时就会将所有列在缓存清单文件中的文件全都缓存至本地。如果是用 Chrome 打开的，就能将其输出至控制台，十分易于理解（图 14.1）。

如果缓存成功，在缓存清单文件中列出这些的文件就能通过保存于本地的应用程序缓存来读取。因此，即使第二次访问时处于离线状态，可能够显示页面内容（图 14.2）。

图 14.1 应用程序缓存的创建

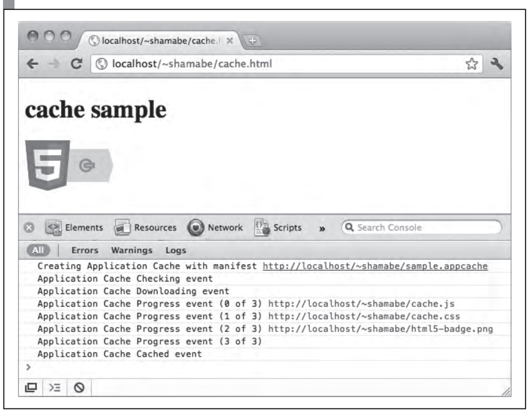
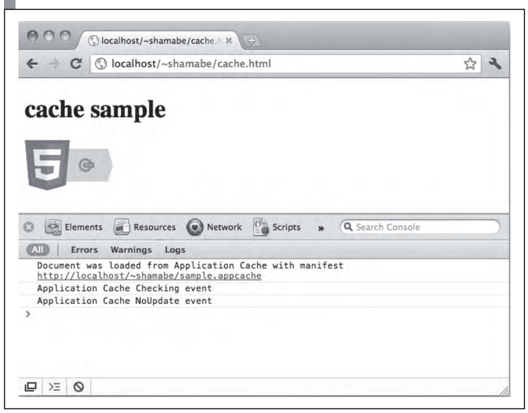


图 14.2 读取应用程序缓存



■ 缓存的更新

在打开注册于应用程序缓存中的页面时，浏览器将首先引用被缓存的文件并显示。之后将在后台自动确认缓存清单文件是否需要更新，在更新之后将会自动重新缓存所有的文件。

也就是说，即使在服务器端更新了文件，在更新后首次访问时浏览器还是会显示旧版本的缓存，对此请加以注意。如果缓存的确认与更新顺利完成，则将从下次访问起引用已更新的新版缓存。

此外，在客户端将会根据缓存清单文件是否被更新来判断是否有必要更新缓存。即如果仅仅是改写了所缓存的文件，已经完成了缓存工作的客户端不会更新缓存。为了更新缓存，必须更新缓存清单文件。

即使缓存规则没有发生更改，也必须对缓存清单文件进行更新以更新缓存。只要通过注释插入版本号或是更新日期，就能够在各种情况下实现对缓存清单文件的更新了（代码清单 14.13）。

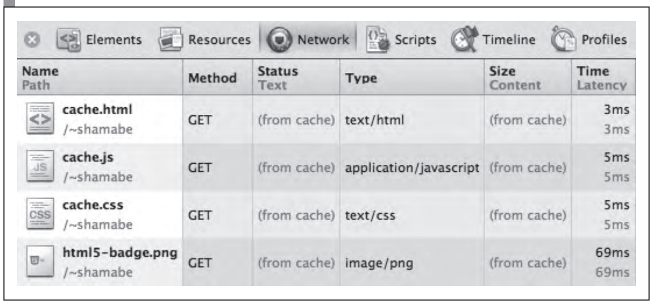
代码清单 14.13　sample.appcache

```
CACHE MANIFEST
# revision 2

CACHE:
./cache.js
./cache.css
./html5-badge.png
```

在这种情况下让浏览器重载的话，就能自动地刷新所有的缓存文件。不过这是所显示的页面引用的仍是旧版本的缓存，对此请加以注意（图 14.3）。需要在下一次重载之后才能引用最新的文件。如果希望精确地控制缓存更新的时机，则必须使用之后将要说明的 Application API。

图 14.3　应用程序缓存的更新



Name Path	Method	Status Text	Type	Size Content	Time Latency
cache.html /~shamabe	GET	(from cache)	text/html	(from cache)	3ms 3ms
cache.js /~shamabe	GET	(from cache)	application/javascript	(from cache)	5ms 5ms
cache.css /~shamabe	GET	(from cache)	text/css	(from cache)	5ms 5ms
html5-badge.png /~shamabe	GET	(from cache)	image/png	(from cache)	69ms 69ms

■ NETWORK 区段

如果在缓存清单文件中写下 NETWORK:，则从该行起及之后的部分都将被作为 NETWORK 区段。在 NETWORK 区段写出的资源将不会被缓存，而始终通过网络访问。而且写于 NETWORK 区段的 URL 将会进行向前一致性比较，于是可以像下面这样仅通过一行来实现多个资源的指定。

```
NETWORK:
/api/
```

NETWORK 区段还有一个重要的作用。对于使用了缓存清单文件的应用程序来说，仅能访问写在 NETWORK 区段中的外部域内的资源。例如在使用 Yahoo!JAPAN 的搜索 API 时，必须像下面这样在 NETWORK 区段中显式地指定白名单。

```
NETWORK:
http://search.yahooapis.jp/
```

由于可以通过域执行访问控制所以通信是很安全的，不过对于一些应用程序来说，可能难以事先指定所有的外部资源。这时可以在 NETWORK 区段使用通配符，以允许所有对外部资源的访问。

```
NETWORK:
*
```

■ FALLBACK 区段

如果在缓存清单文件中写下 FALLBACK:，则从该行起及之后的部分都将被作为 FALLBACK 区段。在 FALLBACK 区段中，我们可以指定某一资源无法被访问时的替代资源。与 NETWORK 区段相同，这时也会对 URL 进行向前一致性比较。

像下面这样进行指定之后，notfound.html 就将被保存在应用程序缓存中。在无法找到资源，或是由于离线而无法连接时，如果该资源在缓存清单文件中的相对路径是以 contents/ 开始的话，就会将 notfound.html 作为替代资源显示。

```
FALLBACK:
contents/          notfound.html
```

14.2.3 ApplicationCache API

通过 ApplicationCache API 能够实现对缓存更新时机更为精确的控制。

在通常情况下，将在开启页面时执行缓存的更新确认与缓存更新，并在下次开启页面时才会反映最新的缓存。因此，必须进行两次重载，最新的缓存才能得以反映。而通过 ApplicationCache API 在合适的时机检查更新的话，就能够将这一问题控制在最小范围内。

可以通过在 window 对象中定义的 applicationCache 对象来使用 ApplicationCache API 的各种功能。接下来，本节将说明使用 applicationCache 来确认缓存更新及反映更新的方法。

■ 对缓存更新的确认

我们可以通过 applicationCache.update 方法在启动页面以外的任意时刻执行对缓存更新的确认。在执行了 update 之后将会确认缓存清单文件的更新，如果文件被更新，则会自动重新缓存所有的文件。

我们可以在用户按下了更新键、从服务器收到了推送通知，或计时器经过了一定的时间间隔等情况之后执行 update。代码清单 14.14 使用了计时器以在每经过一段时间后确认更新。

代码清单 14.14 对缓存更新的确认

```
window.onload = function() {
    // 每小时对更新进行确认
    setInterval(function() {
        applicationCache.update();
    }, 1000 * 60 * 60)
};
```

我们可以通过引用 applicationCache.status 或实现相应的事件处理程序，来获知是否进行了更新或缓存的下载状况。HTML5 相关的异步 API 大都像这样定义了用于状态确认的属性以及事件处理程序，不过在大部分情况下，都只需要对事件处理程序进行实现就可以了。表 14.3 总结了可以通过 applicationCache.status 获取的值，表 14.4 总结了 applicationCache 所能使用的事件处理程序。

表 14.3 applicationCache 的常量属性一览

属性名	整数值	说明
UNCACHED	0	没有应用程序缓存
IDLE	1	正在使用直至上次确认时最新的缓存文件
CHECKING	2	正在确认缓存清单文件的更新
DOWNLOADING	3	正在下载最新的缓存文件
UPDATEREADY	4	已经完成了最新的缓存文件的使用准备
OBSOLETE	5	缓存清单文件已被删除

表 14.4 applicationCache 所能使用的事件处理程序

事件处理程序	说明
onchecking	在开始确认缓存清单文件时被执行
onnoupdate	在确认缓存清单文件后没有更新的情况下被执行
ondownloading	在开始下载缓存时被执行
onprogress	在缓存的下载过程中定期被执行。可以通过 event.total 获取所下载文件的总数，而通过 event.loaded 则可以获取已下载的文件数量
oncached	在所有缓存的下载完成时被执行
onupdateready	在下载完成后能够再次调用 update 时被执行。 而且可以通过在此之后调用 swapCache 来反映最新的缓存
onobsolete	在缓存清单文件被删除时执行
onerror	在发生了某种错误时被执行

■ 对缓存更新的反映

如果在 updateready 事件发生后执行 applicationCache.swapCache 方法，则可以在后台将缓存替换为最新版。不过，当前显示的页面已经引用了的资源则将继续使用之前的版本。

对于基于网页且缓存了大量静态 HTML 的应用程序来说，swapCache 或许是个不错的方法，但如果包含了 JavaScript 逻辑等内容的更新，则只有重载正在显示的页面才有意义。不过，在用户进行操作时突然重载页面并不合适。通知用户现在有了可用的新版本或许是一种稳妥的解决方式。

可以像代码清单 14.15 这样对此进行简单的实现。

代码清单 14.15　通知有新的可用版本

```
applicationCache.onupdateready = function() {
    var ok = confirm('最新版本已经可用。\n' +
                     '要重载页面吗？');
    if (ok) location.reload();
};
```

14.2.4　在线与离线

借助应用程序缓存，程序可以在离线时也能浏览所缓存的信息。然而，对于诸如文档编辑或邮件发送等数据更新类的功能，仅靠这种方式还无法实现离线使用。

而如果将离线时进行的数据更新类操作保存至浏览器所能用的数据库（第 16 章），并在转为在线状态时与服务器进行同步的话，就能够实现离线状态下的数据更新类功能。

在实际实现过程中，需要考虑同步顺序及重试处理等很多问题，不过，比起所付出的这些成本，能够离线使用这些功能魅力更大。有自信处理好这些问题的读者请务必挑战一下。

本节之后将介绍通过程序对网络连接状态进行确认的方法。这在实现离线支持的过程中是不可缺少的。

可以通过引用 navigator.onLine 来获知网络连接状态。还可以通过 online/offline 事件来侦听连接状态的切换时机。online/offline 事件是由 document.body 触发的，并将传递给 document 对象与 window 对象。不过在有些浏览器中，window 对象由于事件处理程序的兼容性问题等而无法正常执行功能，请对此加以注意。

代码清单 14.16 介绍的是一个能够对网络连接状态进行通知的范例。

代码清单 14.16　网络连接状态的通知

```
<p>The network is: <span id="indicator">(state unknown)</span></p>
<script>
// 网络连接状态的更新
function updateIndicator() {
    var indicator = document.getElementById('indicator');
    indicator.textContext = navigator.onLine ? 'online' : 'offline';
}

// 设定 body 的各个事件处理程序
document.body.onload = updateIndicator;
document.body.ononline = updateIndicator;
document.body.onoffline = updateIndicator;
</script>
```

第 15 章 与桌面应用的协作

本章介绍 Drag Drop API 与 File API。这两个 API 都有着独具魅力的功能，如果两者结合使用，就能够实现非常强大的与桌面应用程序的协作功能。

15.1 Drag Drop API

15.1.1 Drag Drop API 的定义

Drag Drop API[1] 是一种能够在浏览器中实现 DOM 元素的拖动与释放操作的 API。拖动与释放功能非常重要，它可以使 Web 应用程序具有接近原生桌面程序的易用性。众所周知，拖动与释放这一功能，其实在很久以前就已经在浏览器中得以实现。

那么，和过去的拖动操作相比，这一 API 究竟有哪些不同之处呢？

■ 实现方式上的区别

过去实现拖动与释放操作的基本方式基于下面的三个流程。其想法本身非常简单，不过，由于从鼠标移动开始，一直到更新 DOM 元素的显示为止，都需要自己管理，因此在实际使用时非常麻烦。

- 通过 mousedown 事件来捕捉 DOM 元素
- 通过 mousemove 事件来移动 DOM 元素
- 通过 mouseup 事件来释放 DOM 元素

而借助于 Drag Drop API，通过 dragstart 及 drop 等新添加的高度抽象的事件，就能够实现更为直观的拖动与释放操作。同时，由于拖动过程中基本的显示更新处理也都交由浏览器来进行，从而使开发者能够将精力集中于程序的开发，以实现运用了这一拖动操作的应用程序。

■ 功能上的区别

当前，人们已经开发了大量支持拖动与释放操作的库。只要利用这些库，就能够很轻松地将跨浏览器支持的拖动与释放功能嵌入应用程序之内。在这种情况下，仍要坚持使用 Drag Drop API 的意义是什么呢？

答案就在 DataTransfer 中。DataTransfer 是对拖动操作中数据的接受与传递提供支持的 API。值得一提的是，通过 DataTransfer 传送数据有一些重要的优点。例如，数据的发送方（拖动起始处）与数据的接收方（释放处）并未限定于同一窗口内。

举例来说，可以由此实现将浏览器中的 DOM 元素拖向文本编辑器，或者将桌面上的文件拖向浏览器等操作。Drag Drop API 消除了 Web 应用程序与原生应用程序之间的界限，是一种非常重要而充满魅力的功能。

---

① http://dev.w3.org/html5/spec/dnd.html

15.1.2　接口

■ 拖动事件

由 Drag Drop API 进行拖动与释放时，数据的发送方（拖动的元素）与接收方（释放的区域）这两者之间是一种松耦合的实现方式。通过对拖动的元素与释放的区域分别实现必要的事件处理程序，就能够完成拖动操作。

表 15.1 列出了能够设定拖动元素的事件处理程序，表 15.2 列出了能够设定释放区域的事件处理程序。

表 15.1　能够设定拖动元素的事件处理程序

事件名	说明
dragstart	在拖动操作开始时被触发
drag	在拖动操作过程中被定期触发
dragend	在拖动操作结束时被触发

表 15.2　能够设定释放区域的事件处理程序

事件名	说明
dragenter	在拖动操作过程中，进入 DOM 元素的领域内时被触发
dragover	在拖动操作过程中，处于 DOM 元素的领域内时被定期触发
dragleave	在拖动操作过程中，离开 DOM 元素的领域时被触发
drop	在 DOM 元素上释放数据时被触发

这些拖动事件继承了鼠标事件的接口，因此也可以通过 screenX 及 clientX 等鼠标事件的属性来确认拖动过程中的位置。曾经使用鼠标事件来实现拖动与释放功能的人，应该已经对相应事件的使用方法有一个大概的理解了吧。

由于拖动事件将会根据拖动操作的状态和合适的时机被触发，因此不必由自己来管理与拖动释放相关的复杂的旗标。例如，drag 事件及 dragover 事件被限定于仅会在拖动操作中被触发。它们与 mouseover 事件不同，即使鼠标没有处于移动状态，也会被定期触发。

各个事件处理程序将会接收以 DataTransfer 形式保存的数据，并对 UI 显示的更新功能进行实现。在使用 Drag Drop API 进行拖动操作时，被拖动元素的捉取图像默认将会随着鼠标的移动而显示相应的内容，而拖动过程中的页面滚动等处理也都将由浏览器来解决。因此，如果没有特别的需求，在最初不考虑拖动中的 UI 显示处理也不会有什么问题。

■ DataTransfer

DataTransfer 是 Drag Drop API 中的核心部分。在所有的拖动事件的事件对象中，都含有 dataTransfer 属性。DataTransfer 最为重要的功能是接收数据，但同时也具有一些其他功能。

- **数据的接收**
- **数据处理方式的指定**
- **拖动图像的设定**

表 15.3 总结了 DataTransfer 的接口。有很多属性只能在特定的拖动事件内被调用或修改，对此请加以注意。本节之后将说明各个属性的详细信息。

表 15.3　DataTransfer 的接口一览

属性名	说明
setData(format, data)	以 format 所指定的格式添加数据（在 dragstart 事件中有效）
getData(format)	以 format 所指定的格式获取数据（在 drop 事件中有效）

（续）

属性名	说明
clearData([format])	以 format 所指定的格式清除数据。如果没有指定 format，则清除所有的数据
types	包含正在拖动的数据的 fromat 的数组
files	包含正在拖动的文件的 File 对象的数组
setDragImage(element, x, y)	设定拖动图像（在 dragstart 事件中有效）
addElement(element)	设定拖动图像（在 dragstart 事件中有效）
effectAllowed	设定允许用于拖动操作的目标的效果。通常会在 dragstart 事件中设定
dropEffect	释放操作的目标的效果，或由用户选择的效果。可以在最新的 dragover 或 dragenter 事件中设定。如果没有设定为特定的值，则会使用标准的操作系统修饰键，在可供选择的效果中进行选择。系统将会在 copy、move、link 与 none 之中选择，并根据选中的效果显示相应的拖动图像

15.1.3 基本的拖动与释放

只要使用 dataTransfer 与最低限度所必需的事件处理程序，就能够实现简单的能够接收数据的范例程序。请通过这一范例来掌握基本的处理流程。

■ 拖动元素的设定

为了使元素能够被拖动，首先要做一些事前的准备处理。要让特定的元素支持被拖动，需要将元素的 draggable 属性设置为 true。

```
<ul>
    <li draggable="true">Seiichiro INOUE</li>
    <li draggable="true">Shota HAMABE</li>
    <li draggable="true">Takuro TSUCHIE</li>
</ul>
```

draggable 属性的值可以被指定为 true、false 与 auto 中的任意一种。如果指定为 auto，则该元素将会使用默认值。例如，img 元素与 a 元素是默认能够拖动的元素。而 li 元素是默认无法被拖动的，因此在这里显式地将 draggable 的属性设定为 true。

■ 拖动方的设定

拖动方（数据的发送方）需要在开始拖动时将数据设置于 dataTransfer 中。可以调用 setData 方法来将所拖动的数据设置给 dataTransfer。setData 是一种只能够在 dragstart 事件处理程序中执行的方法。代码清单 15.1 是一个例子。

setData 的第 1 个参数用于指定数据的格式（MIME Type）。我们可以对 1 次拖动操作指定多个格式的数据。虽然从标准上来说任何格式都是能够被指定的，但实际的实现情况会根据浏览器的不同而有所不同。在代码清单 15.1 的例子中所指定的 MIME Type，至少在最新版的主流浏览器中都是被支持的。

代码清单 15.1 对所拖动数据的设置

```
var element = document.getElementsByTagName('li');

for (var i = 0; i < elements.length; i++) {
    // 在开始拖动时将数据设置于 dataTransfer 中
    elements[i].ondragstart = function(e) {
        // 设置文本数据
        e.dataTransfer.setData('text/plain', e.target.textContent);
        // 设置 HTML 数据
        e.dataTransfer.setData('text/plain', e.target.outerHTML);
        // 设置 URL 数据
        e.dataTransfer.setData('text/uri-list', document.location.href);
    };
}
```

■ 释放方的设定

接下来，我们来对释放方（数据的接收方）进行实现。可以调用 getData 方法来从 dataTransfer 中获取被拖动的数据。getData 是一种只能够在 drop 事件处理程序中执行的方法。

在代码清单 15.2 的范例中，释放区域仅会在被拖动的 dataTransfer 所包含的是文本数据的情况下才允许释放操作，并将通过 alert 语句显示被拖动的文本数据。

代码清单 15.2　获取被拖动的数据

```
<div id="drophere">Drop Here</div>

<script>
// 释放区域
var drophere = document.getElementById('drophere');

// 当拖动元素在释放区域之上时
drophere.ondragover = function(event) {
    for (var i = 0; i < event.dataTransfer.types.length; i++) {
        if (event.dataTransfer.types[i] === 'text/plain') {
            // 取消浏览器的默认操作
            event.preventDefault();
            break;
        }
    }
};

// 当拖动元素被释放于释放区域中时
drophere.ondrop = function(event) {
    // 取消浏览器的默认操作
    event.preventDefault();

    // 获取所拖动的数据
    var yourName = event.dataTransfer.getData('text/plain');
    alert('Hello, ' + yourName + '!');
};
</script>
```

preventDefault 是一个用于取消浏览器的默认操作的方法。在代码清单 15.2 的例子中，有两处调用了 preventDefault 方法。

在 dragover 事件中，浏览器的默认操作是将 drop 事件取消。因此，如果要使 drop 事件生效，则必须在 dragover 事件中调用 preventDefault 以取消默认操作。在这一范例中，只有在 dataTransfer 含有的是 text/plain 格式的数据时，才会执行 preventDefault 而使 drop 事件能够发挥效果。

此外，在 drop 事件内也调用了 preventDefault。如果将链接或文件拖入浏览器，浏览器就将会自动尝试打开它们。这个功能虽然方便，但如果要对所拖入的数据进行自定义处理，浏览器反而是做了多余的操作。为了避免这个问题要使用 preventDefault 来取消这些操作。

15.1.4　自定义显示

如果使用 Drag Drop API，则可以在实现拖动与释放操作时完全不考虑拖动中的 UI 显示问题。不过话虽如此，在拖动与释放操作中，UI 的显示也是很重要的部分。本节接下来将介绍根据需要来显示自定义内容的方法。

■ 拖动图像的更改

在拖动过程中显示的图像（拖动图像）默认将使用拖动元素的捉取图像。可以通过 setDragImage 或 addelement 来更改拖动图像。

setDragImage 与 addElement 是仅能在 dragstart 事件中调用的方法。这两个方法都可以以任意 DOM

元素为参数，将所指定的 DOM 元素的捉取图像作为拖动图像来使用。不过，如果指定的 DOM 元素是 img 元素，则不会使用其捉取图像，而会使用 img 元素的 src 属性所指定的图像。

setDragImage 与 addElement 的区别在于拖动图像的显示位置不同。setDragImage 以拖动图像的左上角为拖动位置来显示该图像，并且可以通过第 2 及第 3 个参数来指定 xy 坐标，以调整显示位置。而 addElement 则将参数所指定的 DOM 元素的当前位置直接作为拖动图像的初始显示位置。

对于诸如日历那样的弹窗式小工具，可以通过 addElement 来移动其显示位置。代码清单 15.3 是一个使用了 addElement 的例子。对于该例这样的情况，如果不通过 addElement 将整个容器指定为拖动图像，则会默认以 handler 作为拖动图像，从而导致显示错乱。

代码清单 15.3 dataTransfer.addElement 的使用示例

```
<div id="container">
    <div id="handler">handler</div>
    ...
</div>

<script>
var container = document.getElementById('container'),
    handler = document.getElementById('handler');

// handler 的拖动开始
handler.ondragstart = function(event) {
    // 将 container 的捉取图像指定为拖动图像
    event.dataTransfer.addElement(container);
};
</script>
```

和 addElement 相比，setDragImage 的使用频率可能会比较低，不过通过 setDragImage 方法，同样能够将自己准备的图像设定为拖动图像。但若希望能将任意图像指定为拖动图像，则还需要进行一些处理。为此，需要使用的是 img 元素。

如果参数被指定为了 img 元素，将不会使用其捉取图像来作为拖动图像，而会使用其 src 属性所指定的图像。因此，可以将 img 元素的 src 属性指定为任意图像之后，再将该元素设定为 setDragImage 的参数，以实现将任意的图像设为拖动图像。

代码清单 15.4 是一个将任意图像设定为拖动图像的示例。在这个例子中，我们对拖动图像的显示位置进行了调整，拖动位置即为拖动图像的中心。

代码清单 15.4 dataTransfer.setDragImage 的使用示例

```
<img id="dragimage" src="drag.gif" style="visibility:hidden; position:absolute;">
<div id="dragme">Drag Me</div>

<script>
document.getElementById('dragme').ondragstart = function(event) {
    var dragimage = document.getElementById('dragimage'),
         offsetX = dragimage.offsetWidth / 2,
         offsetY = dragimage.offsetHeight / 2;

    // 指定拖动图像
    event.dataTransfer.setDragImage(dragimage, offsetX, offsetY);
};
</script>
```

■ 使用 CSS 对拖动图像进行自定义

如果能够通过 CSS 来微调拖动图像所指定的 DOM 元素的颜色及透明度等属性，将是一件非常方便的事。然而可惜的是，目前各个浏览器中与拖动相关的 CSS 功能的统一尚无进展。

在 WebKit 类型的浏览器中，可以通过 -webkit-drag 这一伪类来实现基于 CSS 的拖动图像自定义。下

面是一个例子。

```
/* 设定拖动图像的样式 */
#dragme:-webkit-drag {
        opacity: 0.5;
        -webkit-transform: scale(0.8);
}
```

■ 释放区域的强调显示

通过向用户强调显示应该在何处释放元素，就能够大幅提高拖动操作的易用性。诸如将 dragover 中的元素的背景色改得显眼一些，或者进行元素替换时在插入位置加上辅助线等，都是一些常见的例子。以下是在实现这些效果时的要点。

- 在 dragover 事件中添加效果。
- 不过如果需要根据释放的位置及时机来改变动作，则需要在 dragover 事件中添加能够对动作进行说明的效果。
- 在 dragleave 事件以及 drop 事件中删除效果。

由于在执行释放操作时不会发生 dragleave 事件，因此必须同时在 dragleave 事件和 drop 事件中进行效果的删除处理。由此可以写出代码清单 15.5 中的范例，在这个范例中，dragover 中的元素被赋予了 dragover 这一类名。由于有了代码清单 15.5 中实现的功能，之后可以使用 CSS 来对释放区域进行高度的自定义处理。

代码清单 15.5　在 dragover 的元素中添加类

```
element.ondragenter = function(event) {
    // 效果的添加
    element.classList.add('dragover');
};
element.ondragleave = function(event) {
    // 效果的删除
    element.classList.remove('dragover');
};
element.ondrop = function(event) {
    // 效果的删除
    element.classList.remove('dragover');
};
```

15.1.5　文件的 Drag-In/ Drag-Out

■ 获取桌面程序中的文件

可以借助 dataTransfer 的 files 属性以通过拖动操作来获取桌面程序中的文件。尽管要获取的是一个文件，但也并非难事，只需和通常情况一样，事先在释放区域的 dragover 事件中执行 preventDefault 即可。代码清单 15.6 是一个具体的实现示例。

代码清单 15.6　通过拖动操作来获取文件

```
element.ondragover = function(event) {
    // 使拖动操作有效
    event.preventDefault();
};

element.ondrop = function(event) {
    if (event.dataTransfer.files.length) {
        alert(' 拖动了文件 ');

    // 获取第 1 个 File 对象
```

```
        var file = event.dataTransfer.files[0];
        console.log(file);
        } else {
            alert('没有拖动文件');
        }

        // 防止浏览器打开文件
        event.preventDefault();
    };
```

我们可以通过引用 files.length 来判断是否拖动了文件。files.length 是被拖动的文件的总数。也就是说，可以 1 次拖动并接收多个文件。可以通过下标来引用各个 File 对象。15.2 节将对 File 对象的详细内容进行介绍。

■ **将文件保存至桌面程序**

本节将介绍通过拖动操作来把文件从浏览器中保存至桌面程序的方法。在执笔本书时，该功能仅被 Google Chrome 作为实验性功能进行了实现。目前还不清楚这一功能今后将会如何发展，不过，这是一个非常有用而充满魅力的功能，因此本书将在此对其做简单介绍。

可以按下面所示的格式，将数据设置于 dataTransfer，以实现通过拖动操作将文件从浏览器中保存至桌面程序。

```
event.dataTransfer.setData('DownloadURL', 'MIMETYPE:文件名:文件URL');
```

通过这个功能，就能够实现与原生应用程序之间的双向文件收发，使在桌面环境中运行的 Web 应用程序的实用性得到飞跃性提高。

代码清单 15.7 中是一个文件下载的实现示例。在这个例子中将 download 这一链接拖动至了桌面，以在桌面上保存该链接的资源。

代码清单 15.7　通过拖动操作实现文件的下载

```
<a href="http://www.example.com/foo.mp3"
    data-downloadurl="audio/mpeg:foo.mp3:http://example.com/foo.mp3"
    class="dragout" draggable="true>download</a>

<a href="http://www.example.com/bar.pdf"
    data-downloadurl="application/pdf:bar.pdf:http://example.com/bar.pdf"
    class="dragout" draggable="true">download</a>

<script>
// 获取所有的 dragout 类
var files = document.querySelectorAll('.dragout');
for (var i = 0, file; file = files[i]; i++) {
    file.addEventListener('dragstart', function(event) {
        // 以 DownloadURL 的形式设置数据
        event.dataTransfer.setData('DownloadURL', this.dataset.downloadurl);
    }, false);
}
</script>
```

专栏

DataTransferItemList

本书通过 dataTransfer 的 setData、getData、clearData、types 与 files 属性来实现了数据的收发传递。而在执笔本书时，dataTransfer 的标准中又添加了 items（DataTransferItemList 实例）这一属性。推荐大家在今后的数据收发传递中使用这个接口。

在执笔本书时，还没有浏览器对 DataTransferItemList 的功能提供支持。不过这是一个重要的功能变更，

表 A 对该接口进行了总结。在这里只是整理了接口的内容，其所含有的功能及使用方法几乎没有变化，因此不必过于担心 [1]。

表 A　DataTransferItemList 的接口一览

接口	含义
items	DataTransferItemList 实例
items.add(data, format)	添加数据
items.add(data)	添加 File 对象
items.length	所添加的数据总数的引用
items[index]	所添加的数据的引用
items[index].kind	数据类型的引用（"string" 或 "file"）
items[index].type	数据格式的引用
items[index].getAsString(callback)	以字符串形式获取数据的内容
items[index].getAsFile()	以 File 对象的形式获取数据
delete items[index]	删除所添加的数据
items.clear()	删除所有已添加的数据

15.2 File API

15.2.1　File API 的定义

File API[2] 是一种用于获取在本地保存的文件的信息与内容的 API。在 File API 出现之前，我们虽然可以选择本地文件并发送至服务器，但却无法直接通过 JavaScript 读取文件的信息及内容以进行处理。

此外，虽然在本书中不会过多涉及，但还有 File API: Writer[3] 与 File API: Directories and System[4] 这两个分别用于文件的写入与文件夹结构管理的 API。目前 W3C 正在对它们的标准进行讨论。[5]

由于具备了这些接口，此前那些只能由原生应用程序实现的用于编辑本地文件与管理的程序也终于能够通过浏览器来实现了。

15.2.2　File 对象

■ 文件的选择

通过引用 File 对象就能够获取文件的信息。为了获取在桌面程序中保存的文件的 File 对象，必须让用户显式地选择文件。可以以两种方法来让用户选择文件。

- **通过拖动与释放功能进行选择**
- **通过文件选择对话框进行选择**

通过拖动与释放操作选择的方法，已经在 15.1 节进行了介绍，在此将仅介绍通过文件选择对话框进行选择的方法。在将 input 元素的 type 属性指定为 "file" 之后，就能够使用操作系统提供的标准的文件选择对话框。

如果要改变对话框的行为，则可以指定 input 元素的 accept 属性与 multiple 属性。通过引用 input 元

① 可以通过 Modernizr（http://modernizr.com/）来检查浏览器是否对 HTML5 特性提供了支持。——译者注

② http://www.w3.org/TR/FileAPI/

③ http://www.w3.org/TR/file-writer-api/

④ http://www.w3.org/TR/file-system-api/

⑤ 事实上，这两个 API 的标准已经基本确定，现在已经可以在开发中安心地使用它们了。——译者注

素的 files 属性就能够获取在对话框中选择的文件的 File 对象。如果希望在文件被选中时就开始处理，则可以侦听 input 元素的 change 事件（表 15.4）。

表 15.4　<input type="file"> 的属性一览

属性名	说明
accept	以 MIME Type 来指定允许选择的文件类型。 可以通过逗号分隔符来同时指定多种文件类型
multiple	允许同时选择多个文件
files	含有所选择文件的 File 对象的数组
onchange	在文件被选择时将被执行的事件处理程序

代码清单 15.8 是一个例子，在该范例中，将会显示图像专用的文件选择对话框，并能显示所选择文件的名称。

代码清单 15.8　图像文件选择对话框的实现示例

```
<input type="file" accept="image/*" id="selectFile">
<script>
document.getElementById('selectFile').onchange = function(event) {
    // 获取所选图像的 File 对象
    var file = event.target.files[0];

    // 获取文件的信息
    alert(file.name + ' 已被选择 ');
}
</script>
```

在代码清单 15.8 中，accept 属性被指定为了 "image/*"。在进行文件选择时，常常会希望仅将范围限定至某种媒体内容，对于这种情况，可以以 audio/*、video/*、image/* 这样的别名的方式来指定 accept 属性。而如果希望进一步限定允许选择的图像文件，则可以通过逗号分隔符，以 "image/png, image/gif" 的形式来指定 accept 属性所允许的 MIME Type。

在选择了文件之后就将会执行 onchange 事件处理程序，取得所选文件的 File 对象。在这个例子中，对 File 对象的文件名进行了引用并以提示的形式显示。表 15.5 总结了 File 对象的接口，之后还将对 slice 方法进行说明。

表 15.5　File 对象的接口

属性名	说明
name	文件名
size	文件尺寸（单位 byte）
type	文件类型（MIME Type）
lastModifiedData	文件的最后更新时间
slice(start, end, contentType)	切取文件的一部分

专栏

关于 <input type="file"> 中的 value

<input type="file"> 中的 value 的值将是所选文件的名称。不过根据浏览器的不同，具体情况也会有所差异。

在过去的一些浏览器中，可以通过 value 来获取文件的完整路径，但在最近的浏览器中，出于安全性的考虑而无法获取完整路径。此外，有些浏览器为了提供向下兼容性，而会在路径头部添加 "C:\fakepath\"。

由于这些原因，通常不会引用 <input type="file"> 的 value。如果只是希望获取文件名，请引用 File 对象的 name 属性。

15.2.3　FileReader

■ 接口

通过 FileReader 就可以读取文件的内容（数据）。可以像下面这样创建 FileReader 的实例来使用。

```
var reader = new FileReader();
```

在 FileReader 中根据读取数据的形式而准备了 4 种文件读取方法。可以将这些方法的参数指定为 Blob 对象，以实现异步的文件读取。表 15.6 总结了 FileReader 所具有的方法。

表 15.6　FileReader 的方法一览

方法	说明
readAsArrayBuffer(blob)	以 ArrayBuffer 的形式读取文件
readAsBinaryString(blob)	以二进制字符串的形式读取文件
readAsText(blob [, encoding])	以文本形式读取文件
readAsDataURL(blob)	以 DataURL 的形式读取文件
abort()	中止读取

参数中所指定的 Blob（Binary Large Object）是一种可以高效处理大量数据的接口。File 对象继承了 Blob 的接口，因此可以直接将这些方法的参数指定为 File 对象。

由于文件的读取处理是异步的，因此需要对事件的侦听进行实现，以在读取中以及读取完成后执行所需的处理。表 15.7 总结了 FileReader 所具有的事件处理程序。

表 15.7　FileReader 的事件处理程序一览

事件	说明
onloadstart	在读取开始时被执行的事件处理程序
onprogress	在读取过程中被定期执行的事件处理程序
onload	在读取成功时被执行的事件处理程序
onerror	在读取失败时被执行的事件处理程序
onabort	在读取中止时被执行的事件处理程序
onloadend	在读取结束时被执行的事件处理程序（无论是成功还是失败）

只要设定合适的事件处理程序，就能够在读取状态发生变化后立即进行处理。如果希望能在任意的时刻进行处理，则可以引用 readyState 属性，由自己来检查读取的状态。

在文件的读取结束之后，将会把读取的结果保存于 result 属性中。不过，如果文件的读取失败，result 属性就将会是 null，而错误信息则会被保存于 error 属性之中。

表 15.8 总结了 FileReader 所具有的属性。

表 15.8　FileReader 的属性一览

属性	说明
result	保存了读取的结果
error	保存了读取失败时的错误信息
readyState	表示读取处理状态的整数值
EMPTY	readyState 可以取得的常量（值为 0）。表示读取开始前
LOADING	readyState 可以取得的常量（值为 1）。表示读取中
DONE	readyState 可以取得的常量（值为 2）。表示读取结束（无论正常结束还是出错）

■ 文本文件的读取

如果要一下子介绍所有的功能，将会使说明变得很复杂。我们在此先介绍一个简单的例子。代码清单 15.9 中所写的是为了读取文本文件所必需的最低限度的代码，仅通过 JavaScript 就能以如此简单的方式实现文件的读取。不过需要说明的是，这里省略了错误处理等部分。

代码清单 15.9 文本文件的读取

```
// 以文本文件的形式读取 File 对象的内容
var reader = new FileReader();
reader.readAsText(file);

// 如果读取成功，则将读取结果以提示的方式显示
reader.onload = function(event) {
    var textData = reader.result;       // 或 event.target.result
    alert(textData);
};
```

■ 对错误的处理

在文件读取失败时，有必要将这一结果通知用户。可以通过 onerror 事件处理程序来捕捉读取的错误。而如果要知道错误的原因，则可以引用 FileReader 的 error.code 属性。

表 15.9 是文件读取错误的一览，代码清单 15.10 是捕捉错误的一个实现示例。

表 15.9 文件读取错误一览

属性名	值	说明
NOT_FOUND_ERR	1	没有找到文件
SECURITY_ERR	2	安全性错误（文件被改写、正在执行大量的读取命令、文件限制了来自 Web 应用程序的访问）
ABORT_ERR	3	文件的读取被中止（使用了 abort 方法等）
NOT_READABLE_ERR	4	没有文件的读取权限
ENCODING_ERR	5	超过了 DataURL 的尺寸限制

代码清单 15.10 文件读取错误的捕捉示例

```
reader.onerror = function() {
    if (reader.error.code === reader.error.NOT_READABLE_ERR) {
        alert('没有文件的读取权限');
    } else if (reader.error.code === reader.error.ABORT_ERR) {
        alert('文件的读取被中止');
    } else {
        alert('文件的读取失败');
    }
};
```

在上面的代码中，简便起见使用了 alert 来通知错误。但是强制加入 alert 将会妨碍 UI 操作，因此，对于现在这种异步进行通知的情况，这并不能说是一种良好的通知方式，对此请加以注意。

■ 读取中的处理

如果文件的读取比较费时，则有必要通知用户现在正在读取。可以通过 progress 事件来获知读取的进度。表 15.10 总结了能从 progress 事件对象中引用的信息。

表 15.10 progress 事件对象的属性一览

属性名	说明
lengthComputable	如果文件的长度能够被计算则为 true，否则为 false
loaded	已经被读取的数据尺寸
total	读取目标的文件尺寸

通过进度栏来通知用户现在正在读取是一种方便的做法，而且同时还能够告知用户读取的进度。代码清单 15.11 是一个通过 progress 事件实现了进度栏的范例。

代码清单 15.11 读取中的进度栏的实现示例

```
<div id="progWrap" style="width:200px; height:30px; background:gray;">
    <div id="progBar" style="width:0, height:30px; background:green;"></div>
```

```
</div>

<script>
function readFile(file) {
    var reader = new FileReader();
    reader.onprogress = function(event) {
        // 如果可以计算文件的长度
        if (event.lengthComputable) {
            // 计算进度并更新进度栏的宽度值
            var loaded = (event.loaded / event.total);
            progBar.style.width = proWrap.offsetWidth * loaded + 'px';
        }
    };
    reader.readAsText(file);
}
</script>
```

■ 读取文件的一部分

读取大容量的文件将会花费很多时间。在诸如仅需要上次所取得的文件的增量的情况下，只读取所指定的增量部分效率更高。即使是需要读取整个文件，如果该文件能够被分成若各部分处理，则只需将其分割读取就可以了，就能不必等待整个文件读取完成而开始处理。

可以通过 File 对象的 slice 方法（在执笔本书时仍是 mozSlice/webkitSlice）来实现文件的部分读取。使用 slice 方法可以指定文件的分割位置，并通过 FileReader 仅读取所指定的部分。slice 方法的返回值是 Blob 对象。由于分割位置是在读取文件之前指定的，因此即使是大容量的文件，也能够实现仅对其中的一部分进行高速的读取。

代码清单 15.12 是一个 slice 的使用示例。在这个例子中，假设存在一个以每次向文件后部添加数据的形式所构成的文件，就像是日志文件那样。那么，只要读取上一次所读取部分的增量就能够提高执行效率。

代码清单 15.12　slice 方法的使用示例

```
// 读取的开始位置
var lastPos = 0;

function getDiff(file) {
    // 从上一次的读取位置起，切取之后的部分
    var blob = file.slice(lastPos, file.size);

    // 保存本次读取的位置
    lastPos = file.size;

    // 读取切去的部分
    var reader = new FileReader();
    reader.onload = function() { /* 进行一些处理 */ };
    reader.readAsText(blob);
}
```

15.2.4　data URL

■ data URL 的定义

在此，我们将连同 readAsDataURL 方法一起，对 data URL 进行说明。data URL 指的是以 data: 类型开始的 URL。通常的 URL 被用于指示网页或图像等资源所在的场所，而通过 data URL 则可以直接将这些资源中所含的数据嵌入 URL 之中。

而且由于 data URL 的处理方式与通常的 URL 是相同的，因此通过诸如将图像转换为 data URL 形式这样的手段，就可以以比在原生数据状态下更为简单的方式实现在 HTML 中对图像进行处理。

```
<img src="data:~~~">
```

```
<div style="background:url(data:~~~)"></div>
```

这时，data URL 在 URL 中包含了数据，因此不会像通常的 URL 那样，产生需要从服务器获取图像的请求。浏览器将会解释该 data URL，并将数据打开。如果页面中含有大量小型图像，只要通过将这些图像以 data URL 的形式嵌入页面内，就能够减少对服务器的请求，以减轻服务器的负载，提高显示速度。

■ data URL 的创建

与通常的 URL 一样，data URL 也是由普通的字符串所组成的，因此可以通过 JavaScript 来自由地组成 data URL。这意味着只需要通过 JavaScript 就可以创建出各种类型的资源，从而进一步提升了 data URL 的魅力。

data URL 的书写格式如下所示。

```
data:[<MIME Type>][;base64],<data>
```

Base64 是一种编码方式，它可以仅通过 64 种英文与数字字符，来将多位字符或二进制数据等内容编码为相应的字符串。例如，如果要以图像这类的二进制数据来创建 data URL，则首先需要通过 Base64 编码，将二进制数据转换为可以作为 URL 使用的字符串。如果省略了 [;base64]，则将会以 URL 编码的 ASCII 字符串来指定 <data>。

我们可以通过 encodeURIComponent 函数来进行 URL 编码，通过 btoa 函数（Binary to ASCII）来进行 Base64 编码。代码清单 15.13 是一个创建 data URL 的示例。代码清单 15.13 中所示的三个例子都将在浏览器中显示 "Hello, world!"。

代码清单 15.13　data URL 的创建示例

```
// 文本数据的创建（通过 URL 编码）
document.location = 'data:,Hello%2C%20world!';

// HTML 数据的创建（通过 URL 编码）
var data = encodeURIComponent('<h1>Hello, world!</h1>');
document.location = 'data:text/html,' + data;

// HTML 数据的创建（通过 Base64 编码）
var data = btoa('<h1>Hello, world!</h1>');
document.location = 'data:text/html;base64,' + data;
```

■ readAsDataURL 方法

FileReader 提供了 readAsDataURL 这一方法，用于以 data URL 的形式来读取文件的内容。由于是以 URL 的形式来处理所读取的文件，因此在 HTML 中对文件进行处理也变得非常简单了。代码清单 15.14 是一个例子，这一范例将拖动至浏览器中的图像设定为了页面的背景图像。

代码清单 15.14　readAsDataURL 方法的使用示例

```
document.body.ondragover = function(event) {
    // 使 drop 有效
    event.preventDefault();
};

document.body.ondrop = function(event) {
    event.preventDefault();

    // 获取被拖动的文件的 File 对象
    var file = event.dataTransfer.files[0];

    // 以 data URL 的形式读取被拖动的文件
    var reader = new FileReader();
    reader.readAsDataURL(file);
```

```
        reader.onload = function() {
            // 获取 data URL
            var dataURL = reader.result;

            // 将 data URL 设定为背景
            document.body.style.background = 'url(' + dataURL + ')';

            // 将 data URL 保存至 localStorage
            localStorage.background = dataURL;
        }
    }

    window.onload = function() {
        if (localStorage.background) {
            document.body.style.background = 'url('+ localStorage.background + ')';
        }
    };
```

从代码清单 15.14 中我们可以看出，可以以与通常的 URL 一样的方式，对通过 readAsDataURL 读取的数据进行处理。而且 data URL 只是单纯的字符串，因此也可以将其直接保存于 localStorage 等处。

data URL 正是这样一种让人感到其无限可能性的技术。只要肯花功夫，就能够在不借助于服务器的情况下实现各种各样有趣的功能。

15.2.5　FileReaderSync

FileReaderSync 是一种用于同步读取文件内容的 API。所谓同步读取，指的是文件读取方法的返回值直接就是读取的结果。与异步读取后再通过事件处理程序对数据进行处理的 FileReader 相比，这种方式实现起来较为简单。

我们可以在 Web Workers（参见第 18 章）的环境下使用 FileReaderSync。在 Worker 内使用的话，就不必担心同步读取大型文件可能会引起 UI 的假死。当然，在 Worker 中也是能够使用 FileReader 的。表 15.11 总结了 FileReaderSync 的接口。

表 15.11　FileReaderSync 的接口一览

接口	说明
readAsArrayBuffer(blob)	以 ArrayBuffer 的形式获取文件的内容
readAsBinaryString(blob)	以二进制数据的形式获取文件的内容
readAsText(blob [, encoding])	以文本形式获取文件的内容
readAsDataURL(blob)	以 DataURL 的形式获取文件的内容

可以获取的数据形式的种类与参数的指定方式都与 FileReader 相同。不过，在调用方法时方法的返回值所返回的就是所读取的数据。如果读取失败，则将抛出异常，这时需要通过 try/catch 来进行处理，或者在 Worker 的创建者中对 Worker 实例的 onerror 事件进行侦听以做出处理。

在代码清单 15.15 的范例中，在主线程中由用户选择的 File 对象将在 Worker 内被接收，由 FileReaderSync 来读取文件内容并进行处理。这里对从 File 对象的接收到文件内容的读取的整个过程进行了实现。值得注意的是，与 FileReader 相比，这种实现方式非常简单。

代码清单 15.15　文件同步读取的示例

```
self.onmessage = function(event) {
    var file = event.data;
    var reader = new FileReaderSync();
    var data = reader.readAsText(file);

    /* 进行一些处理 */
};
```

第 16 章 存储

本章讲解 Web Storage 与 Indexed Database 这两种能够在浏览器中使用的存储技术。由于这些技术的出现，永久数据管理这一过去完全由服务器端执行的工作也能够在客户端得以实现，从而可以根据情况选择合适的实现方式。

16.1 Web Storage

16.1.1 Web Storage 的定义

Web Storage 是一种可以简单地将 JavaScript 所处理的数据永久保存的接口。近年来，出现了 Web Storage 等多种客户端存储技术。于是，可以不用再像过去那样必须通过服务器才能进行数据的读写操作。

由于这些技术很好地去除了与服务器的通信部分，因此人们可以享受到性能的提高、开发手续的削减、离线操作的实现等各个方面的优点。

特别是 Web Storage，它非常容易使用，标准也已经确定，浏览器对其的支持情况也较为完善。从各方面来看，它在 HTML5 相关的 API 中属于是一种能被用于实际服务中的使用门槛很低的 API，且已经被用于很多服务中。它的功能是极具魅力的，对于初次接触 HTML5 的人来说，这是一个很好的切入点。

Web Storage 具有以下这些特征：

- Key-Value 型的简单的存储方式；
- 能够以与普通的 JavaScript 对象相同的方式来进行读写操作；
- （与 Cookie 相比）能够保存大容量的数据。

不过 Web Storage 并没有提供诸如创建用于搜索的下标或进行事务处理等功能。如果需要在客户端进行功能更复杂的数据管理，则要使用 Indexed Database、Web SQL Database 或 File Writer API 等方法。

■ Web Storage 的容量

虽然 Web Storage 的标准中没有限制其可能的保存容量，但大部分的浏览器都是以 5MB 为上限对该功能进行实现的。尽管在一些浏览器中也可以根据用户设定来更改这一上限，不过对于面向一般用户公开的 Web 应用程序，还是应该意识到这一限制。

此外，在 Web Storage 中，为每个源（参见之后的专栏）准备了共享的存储空间。即使是不同的服务，只要它们的源是相同的，就能够共享存储。因此，有时 1 个服务可以使用的容量将不足 5MB，对此请加以注意。

■ localStorage 与 sessionStorage

Web Storage 的实体是在全局对象中定义的 localStorage 与 sessionStorage 这两种对象。只要像通常的对象那样对其属性进行读写，就能使所保存的数据在页面跳转时也不会丢失。

localStorage 与 sessionStorage 的区别在于数据的生存周期。对于在 localStorage 中保存的数据来说，只要没有被显式地删除，即使浏览器或计算机执行了重启，这些数据也不会丢失。而另一方面，在

sessionStorage 中，数据仅能在同一个会话内得以保留。下面简单总结了 sessionStorage 的生存周期[①]。

○共享 sessionStorage 的情况

- 通常的页面跳转时
- 在 iframe 内打开了子页面
- 从奔溃中恢复时
- 重新载入时

○没有共享 sessionStorage 的情况

- 在新窗口或新标签页中打开了页面
- 窗口被关闭后又被重新打开时

专栏

源的定义

源（origin）指的是由协议、主机名与端口号所组成的标识符。在 Web Storage 中保存的数据只能被在同一个源中执行的程序所共享。在不同的源中执行的程序之间不能相互引用其 Web Storage。

这种只有在同源的情况下才允许访问的规则称为同源策略。反之，从是否能在不同的源之间安全地进行访问的角度来看，则会用到 Cross-Origin Resource Sharing（CORS、跨源资源共享）这样的术语。源是一个在 HTML5 相关技术的安全性问题中经常被用到的术语，请对此加以牢记。

```
http://foo.example.com/test.html          // 原页面
http://foo.example.com/test2.html         // 同源
http://foo.example.com/bar/test.html      // 同源
http://bar.example.com/test.html          // 跨源
http://foo.example.com:8080/test.html     // 跨源
https://foo.example.com/test.html         // 跨源
```

16.1.2　基本操作

本节之后将介绍 Web Storage 的基本操作。这里所介绍的范例代码都是使用的 localStorage，而对于 sessionStorage 的情况，操作方法也是完全相同的。

■ 数据的读写

可以通过 setItem 方法将数据保存至 localStorage，并通过调用 getItem 方法来引用数据。此外，Web Storage 也提供了可以对值进行读写的语法糖，其操作方法与通常的对象相同。代码清单 16.1 是一个示例。

代码清单 16.1　数据的保存与引用

```
// 数据的保存。以下 3 行是等价的
localStorage.setItem('foo', 'bar');
localStorage.foo = 'bar';
localStorage['foo'] = 'bar';

// 数据的引用。以下 3 行是等价的
var data = localStorage.getItem('foo');
var data = localStorage.foo;
var data = localStorage['foo'];
```

① 即数据将会在何时丢失。——译者注

如果指定了一个不存在的键并试图引用，则会返回 null 值。而对于通常的对象来说，如果指定了一个不存在的键并试图引用，返回的将是 undefined 值。对这一差别大家需要加以注意[①]。

此外，localStorage 只能够对字符串进行读写。虽然根据 W3C 的标准，Web Storage 是可以保存任意的对象的，不过截至执笔本书时，还没有浏览器对此进行了实现。不过，通过 JSON.stringify 与 JSON.parse 方法，就可以在几乎不费功夫的情况下实现对任意对象的完整保存（代码清单 16.2）。

代码清单 16.2 对字符串以外的数据进行读写

```
// 任意的对象
var obj = { x:1, y:2 };

// 将对象转换为 JSON 字符串并保存
localStorage.foo = JSON.stringify(obj);

// 将所保存的 JSON 字符串还原为对象
var obj2 = JSON.parse(localStorage.foo);
```

■ 数据的删除

可以通过调用 removeItem 方法来删除所保存的值。此外，Web Storage 也提供了可以对值进行删除操作的语法糖，其操作方法与通常的对象相同。代码清单 16.3 是一个示例。

代码清单 16.3 数据的删除

```
// 将 'foo' 这个键所保存的值删除
localStorage.removeItem('foo');
delete localStorage.foo;
delete localStorage['foo'];
```

如果指定的是一个不存在的键，则不会有任何效果。如果希望一次删除所有保存于 localStorage 中的值，则可以调用 clear 方法。

```
localStorage.clear();
```

■ 数据的枚举

可以通过 key 方法与 length 属性来枚举保存于 Web Storage 中所有的数据。其中 length 是用于引用所保存的键的总数的属性，而 key 则是用于引用参数所指定的下标的键的方法。代码清单 16.4 是一个示例。

代码清单 16.4 数据的枚举

```
// 枚举所有被保存的数据
for (var i = 0; i < localStorage.length; i++) {
    var key = localStorage.key(i),
        value = localStorage[key];

    /* 进行一些处理 */
}
```

不过需要注意的是，通过 key 方法返回的键并不一定是保序的。在进行值的添加或删除操作时，浏览器可能会改变 key 的顺序。反过来说，只要没有进行值的添加或删除操作，则将一定会保持原有的顺序。

此外还可以通过 for in 语句来枚举所有的键。不过这种情况下，如果向 Object.prototype 等对象中添加了属性，则会被一起枚举。因此需要通过 hasOwnProperty 方法只引用直接属性。代码清单 16.5 是一个示例。

代码清单 16.5 通过 for in 语句枚举数据

```
for (var key in localStorage) {
```

① 截至执笔本书时，对于在 Chrome 中对一个不存在的键进行引用的情况，如果使用的是 getItem 方式则将返回 null 值，如果使用的是属性访问方式则会返回 undefined 值。这一方面的实现应该会在今后得到很好的统一，不过目前仍应对此多加注意。

```
    // 仅引用直接属性
    if (localStorage.hasOwnProperty(key)) {
        var value = localStorage[key];

        /* 进行一些处理 */
    }
}
```

16.1.3　storage 事件

在某个窗口中更改了 Web Storage 中的数据之后，将会在除了更改数据的窗口之外所有的窗口中触发 storage 事件。通过捕捉该 storage 事件并加以适当的处理，就能够在多个同时打开的窗口之间确保数据的一致性。

例如，通过在新标签页中打开的设定页面来更新存储时，可以通过捕捉并处理 storage 事件以使其他所有标签页都能获知设定的更改并执行 UI 的更新处理，从而避免与存储数据之间产生不一致。表 16.1 是 storage 事件对象的属性一览，而代码清单 16.6 则是 storage 事件的使用示例。

表 16.1　storage 事件对象的属性一览

属性	说明
key	被更新的键名
oldValue	更新前的值
newValue	更新后的值
url	被更新的页面的 URL
storageArea	localStorage 或 sessionStorage

代码清单 16.6　storage 事件的使用示例

```
window.addEventListener('storage', function(event) {
    if (event.key === 'userid') {
        var msg = '你好，' + event.newValue + '先生 / 女士';
        document.getElementById('msg').textContent = msg;
    }
}, false);
```

16.1.4　关于 Cookie

一直以来，说起在浏览器中保存数据的方法，我们通常都会想到利用 Cookie 来实现的方法。在不支持 Web Storage 的浏览器中，则可以通过 Cookie 来实现数据的永久保存。作为参考，代码清单 16.7 介绍了 Cookie 的使用方法。

不过由于 Cookie 有以下特点，因此在实际上很少能够被用作 Web Storage 的替代品。

- 容量上限非常小，只有 4KB，因此无法保存较大的数据
- 向服务器发送请求时 Cookie 将被一起发送
- 常用于保存会话信息等重要的信息

代码清单 16.7　Cookie 的使用方法

```
// 值的保存
document.cookie = 'foo=1';
console.log(document.cookie);           // ->'foo=1'

// 值的保存（具有 1 小时的期限）
document.cookie = 'bar=2; expires=' + new Date(Date.now()+3600000).toGMTString();
console.log(document.cookie);           // ->'foo=1; bar=2'

// 值的删除
document.cookie = 'foo=; expires=' + new Date().toGMTString();
```

```
console.log(document.cookie);          // ->'bar=2'

// 1 小时之后
setTimeout(function() {
    console.log(document.cookie);      // ->"
}, 3600000);
```

16.1.5 命名空间的管理

只要没有显式地删除 localStorage 中的数据，这些数据就不会被重置。因此，如果把它们当作本地变量来使用，而胡乱地添加了过多属性，就将会使今后的管理变得十分困难。如果没有对属性名进行有序管理，就可能会在用户的本地环境中积累下很多不需要的垃圾数据，还有可能会在使用同源的服务时产生属性名的冲突等问题。

对于通过 sessionStorage 就能够实现功能的情况，则最好使用 sessionStorage，这样就能省去对数据进行删除管理的麻烦。而对于使用 localStorage 的情况，为了便于管理，应该尽可能地理清最外层的属性名。如果没有什么特别的原因，建议按照代码清单 16.8 的方式，在使用时为每个服务分别准备一个命名空间。

代码清单 16.8 以服务为单位进行命名空间的管理

```
var SERVICE_NAME = 'SERVICE_NAME',
    storage = null;

// 通过 load 事件读取数据至本地变量
window.onload = function() {
    try {
        storage = JSON.parse(localStorage[SERVICE_NAME] || '{}');
    } catch(e) {
        storage = {};
    }
};

// 通过 beforeunload 事件将数据写入 localStorage
window.onbeforeunload = function() {
    localStorage[SERVICE_NAME] = JSON.stringify(storage);
};
```

通过代码清单 16.8 这样的方式，就能在启动时和结束时自动地同步 localStorage 与本地变量（在本例中即 storage 变量），因此在一般情况下只需要读取本地变量的内容即可。而且与本地变量相比，对 localStorage 的读写速度较慢，因此对于将会频繁地访问存储的应用程序来说，这种方式还可能提高程序的性能。

不过，对于多个标签页之间数据不一致的问题，则必须在合适的时机将本地变量中的数据写入 localStorage 之中以进行同步。这时需要捕获 storage 事件，将在其他标签页中执行的 localStorage 更新操作同步至本地变量。代码清单 16.9 是一个示例。

代码清单 16.9 多个标签页之间的数据同步

```
// 在更改设定时将其写入 localStorage
function setStorage(key, value) {
    storage[key] = value;
    localStorage[SERVICE_NAME] = JSON.stringify(storage);
}

// 将在其他标签页中进行的 localStorage 更改读入本地变量
window.onstorage = function(event) {
    if (event.key === SERVICE_NAME && event.newValue) {
        storage = JSON.parse(event.newValue);
    }
};
```

16.1.6　版本的管理

在 localStorage 的实际使用中，必然会发生需要更改结构描述的情况。然而 localStorage 的数据是保存于客户端的，因此无法像在服务器端的数据库中那样自由地更改数据。如果要使管理有序，可以尝试采用代码清单 16.10 这样的版本管理方式。

代码清单 16.10　localStorage 的版本管理的例子

```
window.onload = function() {
    if (!localStorage.version) {
        // 添加属性
        localStorage.foo = 'foo';
        localStorage.bar = 'bar';

        // 更新版本
        localStorage.version = '1.0';
    }

    if (localStorage.version === '1.0') {
        // 整合单独设置的属性
        localStorage.foobar = JSON.stringify({
            foo: localStorage.foo,
            bar: localStorage.bar
        });
        // 删除属性
        delete localStorage.foo;
        delete localStorage.bar;

        // 更新版本
        localStorage.version = '1.1';
    }
}
```

16.1.7　对 localStorage 的模拟

localStorage 已经在很多浏览器中得到了实现，在各种 HTML5 相关 API 中它是较易正式使用的一种。不过由于 IE6 与 IE7 并不支持 localStorage，因此为了对这些浏览器提供支持必须采取一些对策。

这里所采用的方法是在未提供支持的浏览器的全局作用域中创建一个 localStorage 对象，并模拟 localStorage 中各个方法的功能。通过这种方式，就能够在不更改现有程序的情况下使其能够运行于不支持 localStorage 的浏览器中。这样一来，就避免了使用条件判断语句而使程序变得不必要地复杂。

代码清单 16.11 是一个对 localStorage 进行模拟的实现的示例。

代码清单 16.11　对 localStorage 的模拟

```
window.localStorage = window.localStorage || (function() {
    var storage = {};

    return {
        setItem: function(key, value) {
            storage[key] = value;
        },
        getItem: function(key) {
            return storage[key];
        },
        removeItem: function(key) {
            delete storage[key];
        },
        clear: function() {
            storage = {};
```

```
            },
            emulated: true
        };
    })();
```

代码清单 16.11 的例子并没有实现数据的永久保存功能。本来，使用了 localStorage 的应用程序需要在 localStorage 为空的状态下也能够正常运行，因此，即使存储在访问过程中被重置，运行也不会出错。如有必要，可以同时借助 Cookie 或 Flash 来模拟数据的永久保存功能。

16.2 Indexed Database

16.2.1 Indexed Database 的定义

Indexed Database[①] 是一种在浏览器中通过 JavaScript 进行操作的功能强大的数据库。之前介绍的 Web Storage 虽然具有使用简便的特点，但并没有提供创建用于检索的索引以及事务的功能。Indexed Database 是一种用于在客户端实现更大规模的复杂数据管理的技术，其标准的制订也正在进行中。

如今，Web SQL Database（参见专栏）的标准制订已中止，Indexed Database 便成了可以在客户端使用的功能强大的数据库的唯一选项。其标准制订与浏览器的实现情况的动向也因此广受瞩目。尽管其标准和实现都还没有最终确定，但它确实是一种重要的 API，因此，本书将用一些篇幅来介绍这一概念。所刊的代码在 Chrome14 与 Firefox6 中进行了验证运行。

专栏

关于 Web SQL Database

Web SQL Database（http://www.w3.org/TR/webdatabase/）是一种运行于浏览器中的关系数据库。通过其名称也可以知道，它可以通过 SQL 语句来执行数据的插入及检索等操作。虽然 Web SQL Database 比 Indexed Database 更早地开始进行标准的制定与浏览器的实现，不过由于它的标准和实现十分依赖于 SQLite 这一 SQL 语言的变体，因此也有很多人对此持以质疑。如今，Web SQL Database 的标准制订已经中止。

不过，现在已经能够在 Chrome、Safari 以及 Opera 中使用 Web SQL Database 了。由于 IE 及 Firefox 尚没有对其提供支持，因此我们难以在针对 PC 的服务中使用 Web SQL Database，不过，iOS 及 Android 设备则几乎是得到了完全的支持。移动设备通信线路的速度较慢，因此本地数据库将带来很大的优势。而且目前 Indexed Database 也尚未达到可以实际应用的阶段，因此有不少服务都使用了 Web SQL Database。

16.2.2 基础架构

Indexed Database 和 Web Storage 一样，仅具有在同一个源中执行的程序所共享的空间。在 1 个源所拥有的空间中可以创建多个数据库，而在 1 个数据库中又可以创建多个对象存储（ObjectStore）。

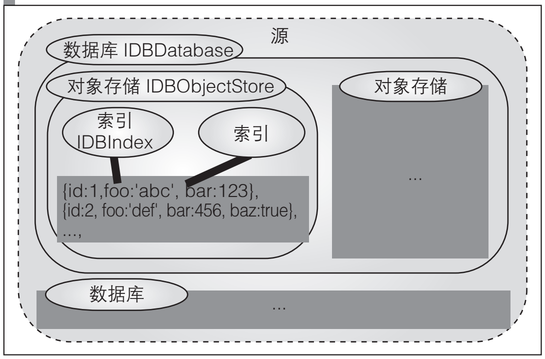
对象存储是一种用于保存数据的容器，在对象存储中可以保存任意类型的 JavaScript 对象。为了便于理解，我们可以认为在关系数据库所说的表对应的就是对象存储，而表的行所对应的就是任意的 JavaScript 对象。

在对象存储中，我们可以为所保存的 JavaScript 对象的任意属性创建索引，且可以同时创建多个索引。通过索引的创建，就能够对相应的属性进行指定范围的高速检索。

图 16.1 总结了当前已经提及的 Indexed Database 的结构组成。

① http://www.w3.org/TR/IndexedDB

图 16.1　Indexed Database 的组成图



16.2.3　连接数据库

我们可以通过调用 indexedDB 对象的 open 方法来连接数据库（IDBDDatabase 实例）。由于连接是以异步的方式执行的，因此可以侦听 open 所返回的请求对象（IDBRequest）的 success 事件与 error 事件。如果连接成功，则将执行 onsuccess 事件处理程序，并能够对数据库进行引用。

在执笔本书时，indexedDB 尚未取消其开发商前缀，因此需要稍作一些事前准备工作（代码清单 16.12）。代码清单 16.13 是从开始直至完成与数据库的连接的范例代码。

代码清单 16.12　对 indexedDB 的开发商前缀进行支持

```
var indexedDB = window.indexedDB ||
                window.webkitIndexedDB ||
                window.mozIndexedDB;
```

代码清单 16.13　连接数据库

```
var db = null;

// 连接数据库
var request = indexedDB.open('testdb');

// 连接数据库成功
request.onsuccess = function(event) {
    // 使之可以通过全局变量 db 引用数据库
    db = event.target.result;
};

// 连接数据库失败
request.onerror = function(event) {
    alert(' 连接数据库失败 ');
};
```

16.2.4　对象存储的创建

在读写数据之前，首先需要创建对象存储这一用于保存数据的容器。我们可以通过调用 createObjectStore 方法来创建对象存储。只能够在数据库的版本更改事务中执行 createObjectStore，对此请加以注意。在调用了 setVersion 方法之后将会自动地在内部开始该事务。代码清单 16.14 是一个示例。

代码清单 16.14　对象存储的创建

```
// 更改 DB 的版本
```

```
var request = db.setVersion('1.0');

request.onsuccess = function(event) {
    // 创建对象存储
    var store = db.createObjectStore('books', {
        keyPath: '_id',
        autoIncrement: true
    });
};
```

createObjectStore 的第 1 个参数所指定的是对象存储的名称。在这里创建的是一个名为 book 的对象存储。其第 2 个参数是与键相关的设定。这里将 keyPath 指定为了 _id，并将 autoIncrement 设为了 true，于是在向对象存储中添加数据时将会自动加上能够自动增加的 _id 属性。

根据键的设定方式不同，添加数据时的行为也会有些差异。依照 autoIncrement 是 true 还是 false，以及 keyPath 是否被指定，总共可以组成 4 种模式。

- **指定了 keyPath 且 autoIncrement 为 true 时**
  在添加数据时将会自动对所指定的属性添加唯一键。
- **指定了 keyPath 且 autoIncrement 为 false 时**
  在添加数据时，所指定的属性必须具有唯一键。
- **没有指定 keyPath 且 autoIncrement 为 true 时**
  在添加数据时，将会自动添加唯一键。该键不被包含在所添加的数据之中，而是另行管理。
- **没有指定 keyPath 且 autoIncrement 为 false 时**
  在添加数据时必须指定唯一键。该键不被包含在所添加的数据之中，而是另行管理。

16.2.5 数据的添加・删除・引用

在通过键对数据进行添加、删除、引用操作时，需要调用各个对象存储的 add（put）、delete，及 get 方法。所有的方法都是异步执行的，返回值是一个请求对象，因此可以侦听这些对象的 success 事件与 error 事件。

代码清单 16.15 是一个实现示例，代码清单 16.16 是一个运行示例。详细说明请参见源代码中的注释。

代码清单 16.15 各种 DB 操作的实现示例

```
// 对开发商前缀提供支持
var IDBTransaction = window.IDBTransaction || window.webkitIdbTransaction;

// 用于数据添加的包装函数
function addData(data) {
    // 事务的开始
    var transaction = db.transaction(['books'], IDBTransaction.READ_WRITE);
    // 添加数据（也可以用 put 来替代 add）
    var request = transaction.objectStore('books').add(data);

    request.onsuccess = function(event) {
        // 如果成功，则返回键
        var key = event.target.result;
        console.log('success! key ->', key);
    };
}

// 用于数据引用的包装函数
function getData(key) {
    // 事务的开始
    var transaction = db.transaction(['books']);
    // 数据的引用
    var request = transaction.objectStore('books').get(key);
```

```
    request.onsuccess = function(event) {
        // 如果成功，则返回数据
        var data = event.target.result;
        console.log('success! data ->', data);
    };
}

// 用于数据删除的包装函数
function deleteData(key) {
    // 事务的开始
    var transaction = db.transaction(['books'], IDBtransaction.READ_WRITE);
    // 数据的删除
    var request = transaction.objectStore('books').delete(key);

    request.onsuccess = function(event) {
        console.log('success');
    };
}
```

代码清单 16.16　各种 DB 操作的运行示例

```
js> addData({ isbn:'477413614X', name:'详解 C#' });
key -> 1

js> addData({ isbn:'4774139904', name:'详解 Java' });
key -> 2

js> addData({ isbn:'4774144371', name:'详解 PHP' });
key -> 3

js> getData(3)
data -> {
    _id: 3,       // 自动添加 _id 属性
    isbn:'4774144371,
    name: '详解 PHP'
}

js> deleteData(3)
success

js> getData(3)
data -> undefined
```

16.2.6　索引的创建

我们可以通过调用 createIndex 方法来创建索引，且对任意的属性都能够创建索引。如果希望对多个属性创建索引，则将依照属性个数来创建索引。

此外，只能在数据库的版本更改事务中执行索引的创建操作，对此请加以注意。代码清单 16.17 是一个索引创建的实现示例。

代码清单 16.17　索引的创建

```
// DB 的版本更改
var request = db.setVersion('1.1');

request.onsuccess = function(event) {
    var transaction = event.target.result;
    var store = transaction.objectStore('books');

    // 创建 isbn 属性的索引
    var index = store.createIndex('isbnIndex', 'isbn');
};
```

16.2.7 数据的检索与更新

本节之后将说明索引的使用方法。通过使用索引，就能够对创建有索引的属性进行高速的指定范围的检索。可以使用 IDBkeyRange 来指定范围。表 16.2 对 IDBKeyRange 的接口进行了总结。

表 16.2 IDBKeyRange 的接口一览

方法	说明
only(value)	创建一个仅含有 value 的范围
lowerBound(value [, open])	创建一个以 value 为下限的范围（如果 open 被指定为了 true，则该范围将不包含 value）
upperBound(value [, open])	创建一个以 value 为上限的范围（如果 open 被指定为了 true，则该范围将不包含 value）
bound(lower,upper[,lowerOpen, upperOpen])	创建一个以 lower 为下限，以 upper 为上限的范围（可以通过 lowerOpen 与 upperOpen 分别来指定是否要在该范围中包含其边界值）

我们可以通过对象存储的 index 方法来引用已有的索引。而通过 openCursor 方法则能根据索引来检索数据。如果将 openCursor 的参数指定为了 IDBkeyRange 对象，则能够依次检索在指定范围内包含的数据，并对符合条件的数据进行引用、更新、删除等处理。

在成功对数据进行了检索时将会触发 success 事件。目前在 Firefox 中，即使没有数据匹配，也将会触发 success 事件，因此有必要通过 if 语句来确认 cursor 是否存在[①]。在调用了 cursor.continue 之后将会继续检索下一个数据，如果数据的检索成功，则将再次触发 success 事件。

代码清单 16.18 是一个通过索引对数据进行检索与更新处理的实现示例。详细的说明请参见代码内所写的注释。

代码清单 16.18 数据的检索与更新

```
// 对开发商前缀提供支持
var IDBTransaction = window.IDBTransaction || window.webkitIDBTransaction;
var IDBKeyRange = window.IDBKeyRange || window.webkitIDBKeyRange;

// 对事务与对象存储的准备
var transaction = db.transaction(['books'], IDBTransaction.READ_WRITE);
var store = transaction.objectStore('books');

// 创建用于指定范围的对象
var range = IDBKeyRange.bound('40000000000', '5000000000');

// 通过索引检索数据
var request = store.index('isbnIndex').openCursor(range);

request.onsuccess = function(event) {
    // 获取 IDBCursor 对象
    var cursor = event.target.result;

    // 如果数据存在
    if (cursor) {
        // 获取数据
        var data = cursor.value;

        // 更新数据
        if (data.name === '详解 Java') {
            // 即使没有对结构描述进行定义，也能够对数据进行扩展
            data.author = '井上诚一郎， 永井雅人， 松山智大';
            cursor.update(data);
        }
```

① 截至翻译本书时仍如此，且今后也不太可能发生变化。关于可能的标准更改，请留意官方文档。——译者注

```
            // 删除数据
            if (data.name === '详解 C#') {
                cursor.delete();
            }

            // 检索下一个数据
            cursor.continue();
    }
};
```

16.2.8　数据的排序

在通过索引检索数据时，可以指定检索顺序并取得数据。如果希望指定检索顺序，可以在 openCursor 方法的第 2 个参数中指定遍历顺序。可以作为遍历顺序进行指定的对象已经被定义为了 IDBCursor 中的常量属性。表 16.3 总结了 IDBCursor 中的常量属性。

表 16.3　IDBCursor 的常量属性一览[①]

属性	整数值	说明
NEXT	0	以升序方式获取（默认值）
NEXT_NO_DUPLICATE	1	以升序方式获取 （如果索引所指定的属性发生了重复，则仅读取第一个数据）
PREV	2	以降序方式获取
PREV_NO_DUPLICATE	3	以降序方式获取 （如果索引所指定的属性发生了重复，则仅读取第一个数据）
ENCODING_ERR	5	超过了 DataURL 的大小限制

16.2.9　事务

我们可以在 Indexed Database 中使用事务功能。之前提到过，在进行数据库的版本更改等处理时，将会在内部自动地创建事务。

在创建了事务的方法已经结束，或开始了新的事务时，该事务将被自动提交。而如果在事务被提交之前就发生了错误，则将自动执行回滚操作。通过在程序中调用事务的 abort 方法可以显式地执行回滚操作。

代码清单 16.19 是一个将所有符合检索条件的数据全部删除的范例。不过，若是在检索结果中含有标有 readonly 旗标的数据，则将取消所有的删除处理。

代码清单 16.19　回滚操作的实现示例

```
var request = index.openCursor(keyRange);
request.onsuccess = function(event) {
    var cursor = event.target.result;
    if (cursor) {
        // 如果标有 readonly 旗标
        if (cursor.value.readonly) {
            var transaction = event.target.transaction;
            // 中断处理并执行回滚操作
            transaction.abort();

        } else {
            // 数据的删除
            cursor.delete();
            // 检索下一个数据
            cursor.continue();
        }
```

① IDBCursor 的常量标准。

```
    }
};
```

16.2.10 同步 API

在 Indexed Database 的标准中制订了一些用于对各种数据库操作进行同步处理的 API。这些 API 只能在 Web Workers（参见第 18 章）的环境下使用。已经进行了说明的那些 API 之所以都能逐一以事件驱动的方式进行处理，是因为它们没有阻塞用户的 UI 操作。而在工作线程内部的话，即使要同步执行复杂的处理，也不用担心 UI 会假死。

可惜的是，在执笔本书时还没有浏览器对 Indexed Database 的同步 API 提供支持。以下简单的范例代码是参考了 W3C 的标准而写成的，可作为参考。值得一提的是，与事件驱动方式的代码相比，其实现相当简单（代码清单 16.20）。

代码清单 16.20 Indexed Database 的同步 API

```
// 连接数据库
var db = indexedDBSync.open('testdb');

if (db.version !== '1.0') {
    // 版本的设定
    var transaction = db.setVersion('1.0');
    // 对象存储的创建
    transaction.createObjectStore('books', {
        keyPath: '_id',
        autoIncrement: true
    });
}

// 数据的添加
var transaction = db.transaction(['books'], function() {
    var store = transaction.objectStore('books');
    store.add({ isbn:'4774139904', name:'详解 Java' });
}, IDBTransactionSync.READ_WRITE);
```

第 17 章 WebSocket

本章将介绍 WebSocket。WebSocket 是一种在浏览器的应用程序中实现高效的双向通信的技术。在很长时间内，通信相关的技术都没有取得显著进展，随着 WebSocket 的出现，开发更为快速、简单的 Web 应用程序成为可能。

17.1 WebSocket 概要

17.1.1 WebSocket 的定义

WebSocket[1] 是一种用于在服务器与客户端之间实现高效的双向通信的机制。最近，在 Gmail 这类重视数据实时性的 Web 应用程序中常常会使用 WebSocket，这种技术因此广受瞩目。如今 JavaScript 的处理性能已经得到了大幅改善，Web 应用程序的性能瓶颈已经转移到了网络通信部分。WebSocket 被认为是一种能够实现实时 Web 功能的关键技术，备受期待。

WebSocket 的 API 组成非常简单。通过 WebSocket，就能够在 1 个 HTTP 连接上自由地双向收发消息。与通过结合使用 XMLHttpRequest 与 Server-sent Events 而实现的双向通信相比，这种方式具有通信效率更高、设计与实现容易等优点。

17.1.2 现有的通信技术

■ XMLHttpRequest

在 XHR（XMLHttpRequest）出现之后，由客户端发往服务器方向的异步通信得以实现。准确地说，在更早之前我们就能够通过 iframe 及 img 等方式来强行实现异步通信了，不过在 XMLHttpRequest 确立了其标准的通信手段的地位之后，Web 应用程序的异步通信领域的技术才得到了飞跃性的进步。

但是，XHR 有一个很大的问题，它无法跨源通信。为了实现跨源通信而出现了 JSONP 等技巧性较强的方法，且至今仍被广泛使用。现在 W3C 为了能通过一种规范的方式来解决跨源通信的问题，正在对 XMLHttpRequest Level 2[2] 的标准进行讨论[3]。

XHR 技术是在过去不具备状态的通信技术的基础上设计的。与 WebSocket 相比，在通过 XHR 进行通信时必须添加请求头部，因此即使只发送 1 个字节的信息，也需要同时发出数 KB 的多余的信息。在诸如聊天等希望每输入 1 个字符便向服务器发送信息时，如果程序很重视其实时性，就可能因此发生性能下降。

■ Server-Sent Events

由服务器发往客户端方向的通信（推送通信）领域中，在很长一段时间内都没有一种能够被称为标准技术的基本技术，因此，一些强行使用了大量技巧性较强的手段得到了广泛运用。如今，W3C 为了能

① http://dev.w3.org/html5/websockets/

② http://www.w3.org/TR/XMLHttpRequest2/

③ 2011 年底，XMLHttpRequestLevel 2 已经并入 XMLHttpRequest 标准。参见 http://www.w3.org/TR/XMLHttpRequest/。——译者注

够以一种优雅的方式来实现来自服务器的推送通信，正在对 Server-Sent Events[①] 的标准进行讨论。

Server-Sent Events 的特点是，只需以标准所指定的格式从服务器端返回通常的 HTTP 响应，就能够实现推送通知的功能，因此可以直接以现有的 HTTP 服务器技术来设计与实现。同时，由于其协议也非常简单，标准也已经确定，因此能够放心地进行使用。

然而，尽管现在的市场对推送通信已经有了巨大的需求，却还是有很多的浏览器没有对 Server-Sent Events 及 WebSocket 提供支持。因此，本节将介绍一些广泛用于过时浏览器的替代技术。

■ AJAX（轮询）

首先可以考虑使用轮询方式，这是一种最为简单的用以实现推送通信的方法。所谓轮询，指的是逐次向服务器确认是否有发送请求的方法。具体说来，它以 AJAX 的方式从客户端定期地向服务器发送请求，确认服务器的状态，并根据服务器的状态，执行合适的操作。代码清单 17.1 是一个轮询的实现示例。

代码清单 17.1 轮询的实现示例

```
// 定期确认服务器的状态
setInterval(function() {
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function() {
        if (xhr.readyState == 4 && xhr.status == 200) {
            var res = JSON.parse(xhr.responseText);
            // 如果有更新，则执行操作
            if (res.serverStateChanged) {
                /* 进行一些处理 */
            }
        }
    };
    xhr.open('GET', 'http://www.foo.org/checkServerState');
    xhr.send();
}, 100);
```

这个例子使用了 setInterval，并通过 XMLHttpRequest 定期确认服务器的更新。这是一种标准技术，能够同时用于客户端与服务器端，而且具有易于实现，运行稳定的优点。但是它存在下面这些问题。

- **即使服务器没有更新，也会发生请求响应，而白白增加了服务器与客户端的负荷。**
- **即使服务器进行了更新，如果客户端没有发来请求，也就无法获知这一情况，因而更新通知需要额外花费一些时间。**

■ Comet（长轮询）

所谓的 Comet 是一种总称，指的是一种只有在必要时才从服务器返回响应的方式。可以通过若干种方法来实现 Comet。这里介绍的是一种被称为长轮询的方法，它可以在轮询的基础上进行少许更改之后实现。

对于通常的 HTTP 通信来说，在收到了客户端发来的请求之后，服务器就会立即返回响应并切断连接。而在长轮询中，在客户端发来请求之后，服务器将会保留响应，并维持连接，因此可以在任意的时间点从服务器返回响应。而客户端在收到了响应的同时，将会再次向服务器建立连接。

代码清单 17.2 是长轮询的实现示例。

代码清单 17.2 长轮询的实现示例

```
function connect() {
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function() {
        if (xhr.readyState == 4) {

            // 来自服务器的推送通知
            if (xhr.status == 200) {
```

① http://dev.w3.org/html5/eventsource/

```
                /* 进行一些处理 */
            }

            //  再次连接
            connect();
        }
    };
    xhr.open('GET', 'http://www.foo.org/comet');
    xhr.send();
}
```

在使用长轮询时，我们必须事先在服务器端进行设定，应将同时连接数及 Keep-Alive 设得大一些。与轮询相比，长轮询避免了一些不必要的通信过程，但也需要在有更新时再次连接。这一点仍然有改进余地。

■ Comet（流）

有一种改进了长轮询的缺点的 Comet 实现方式。具体来说，通过由客户端发出的第一个请求，建立连接，并在维持该链接的同时从服务器不断向客户端返回响应。由于服务器端始终处于在发送响应的状态，因此将这种方式称为流。

为了通过流来实现 PUSH 通信，就必须一边在客户端接收响应，一边分析其内容，并进行合适的处理。事实上，W3C 已经很全面地定义了 Server-Sent Events 这样一种协议，用以实现这种处理方式。

在此，为了能在过去的浏览器中也能够实现流处理，而使用了一种技巧性较强的实现手段。接下来，我们来介绍这种使用了 iframe 的实现示例（代码清单 17.3、代码清单 17.4）。

代码清单 17.3　流的实现示例（客户端页面）

```
<iframe id="iframe_streaming" style="display:none;"></iframe>

<script>
function callback(res) {
    /* 进行一些处理 */
}

function connect() {
    // 通过 iframe 来建立连接
    var iframe = document.getElementById('iframe_streaming');
    iframe.src = 'http://www.foo.org/comet';
}
</script>
```

代码清单 17.4　流的实现示例（服务器响应）

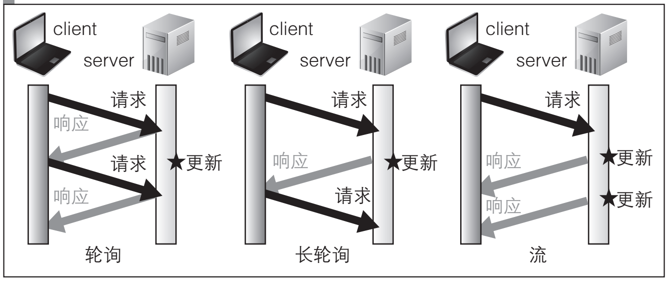
```
<script>window.parent.callback('DATA_1');</script>
<script>window.parent.callback('DATA_2');</script>
...
```

接收 iframe 后将会以通常的 HTML 的形式分析服务器的响应。如果服务器发送了一个 script 标签，该 script 的内容就会在 iframe 内被分析并执行。只要源相同，就能够在 iframe 内调用父页面的 JavaScript 函数，因此，就能在任意时刻在服务器端执行客户端中的任意函数。

这种实现方式的缺点在于，浏览器对 iframe 的实现将会对此造成很大的影响。比如，在使用这种方式时，iframe 将始终处于读取状态，而不会完成读取。因此，根据浏览器的实现方式的不同，可能会在标签中始终显示正在载入的图像。此外，如果不断地发送响应，页面体积将会不断增加。因此有必要在合适的时机刷新 iframe。

图 17.1 对轮询、长轮询及流的执行方式进行了比较。

图 17.1 PUST 通信技术的比较



17.1.3 WebSocket 的标准

WebSocket 的标准现在依然处于制订过程中。WebSocket 的标准可以分为与在客户端使用的 JavaScript API 相关的标准，以及与服务器间的通信相关的协议标准。本章将主要阐述前者，即 JavaScript API 标准，而对之后的协议标准部分，则仅会对一些必要之处进行说明。

WebSocket 的协议标准正在由 IETF 进行制订[①]。在 2010 年 11 月，一个协议方面的重大安全隐患被发现。因此，Firefox4 与 Opera11 默认禁用了对 WebSocket 的支持。现在，正计划对协议标准中的这一安全问题进行处理。Firefox6 对这一修改后的标准进行了实现，从这一版本起，将会再次启用 WebSocket[②]。

如果要在 Firefox4 及 5 中使用 WebSocket，则需要在地址栏中输入 about:config，在所显示的设定页面中，将 network.websocket.override-security-block 一项更改为 true。同样地，在使用 Opera11 的情况下，需要在地址栏中输入 opera:config，在所显示的设定页面中，勾选 Enable WebSocket 这一复选框。

17.1.4 WebSocket 的执行方式

在通过 WebSocket 开始双向通信时，首先需要与服务器建立连接。而用于建立连接的请求，是由客户端通过 HTTP 方式发送的。服务器端将会确认连接对象的源以及协议，并发送连接许可的响应。在发送了响应之后，浏览器将会把该连接升格为 WebSocket。这一连串的流程称为握手。

从客户端 Web 应用开发者的角度来看，在完成握手并建立 WebSocket 连接的过程中，必需的 JavaScript 代码仅有一行。

```
var ws = new WebSocket('ws://www.foo.org/bar', 'subprotocol');
```

在握手通信中，我们通常会使用 GET 请求。而在建立连接过程中所必需的协议（"ws://" 或 "wss://"）、子协议、域、端口等，则都是在握手时指定的。此外，仅在握手时才会在通信中添加 HTTP 头部，因此，有必要在握手时，执行所有使用了 UserAgent 或 Cookie 来进行的用户及设备认证。

在完成了握手之后，就建立了一个连接，可以自由地进行消息收发。在之后的通信过程中，不会产生通信所不需要的头部信息的收发，也不需要进行客户端的认证处理。也就是说，WebSocket 通信是含有状态的，即使仅收到了 1 个字节的数据，也能够确定该发送方客户端。

① http://tools.ietf.org/html/draft-ietf-hybi-thewebsocketprotoco

② 截至翻译本书时，主流浏览器中对 WebSocket 提供了支持的有以下这些：Internet Explorer 10（及移动版）、Mozilla Firefox 6（Firefox for Mobile 7）、Google Chrome 4（及移动版）、Safari 5（及 iOS4.2 之后）、Opera 12.10（及移动版）、BlackBerry 7（需要设定）。而最新的 Android 4.2 的内置浏览器尚未对 WebSocket 提供支持。——译者注

17.2 基本操作

17.2.1 连接的建立

在建立连接时，我们首先需要在客户端执行对 WebSocket 类的构造函数调用，以创建 WebSocket 实例。构造函数调用的第 1 个参数所指定的是，将要进行连接的 WebSocket 服务器的 URL，而其第 2 个参数则需要指定子协议名。

```
var ws = new WebSocket('ws://www.foo.org:8888/bar', 'subprotocol');
```

WebSocket 可以选择 "ws://" 或 "wss://" 这两种协议。如果将协议指定为 wss，就能够以 TLS 对通信加密。如果没有指定端口，则将分别默认使用 80 和 443 端口。此外，在这种情况下，将不会受到同源策略的制约。如有必要，则可以在 WebSocket 服务器端限制连接。

第 2 个参数中的子协议名是可以省略的。反过来，也能够通过数组，来指定多个子协议名。如果指定了子协议，则会在服务器端选择一个将要使用的子协议并返回。子协议是应用层的协议。如果希望应用程序能够根据所选择的子协议，来切换不同的处理操作，可以使用这种方式。

在进行构造函数调用之后，内部将会执行握手处理。一旦建立了连接，WebSocket 实例中就会触发 open 事件，因此，需要像代码清单 17.5 这样，设定事件处理程序，并实现必要的处理操作。例如，在启用发送按钮，或向服务器请求初始数据时，可以将其作为触发器来使用。

代码清单 17.5　连接建立时的事件处理程序

```
ws.onopen = function(event) {
    /* 进行一些处理 */
};
```

17.2.2 消息的收发

在建立了连接之后，就可以进行消息收发了。如果要将消息发送至服务器，则需要将希望发送的数据传递至 send 方法的参数。如果要从服务器接收消息，则可以通过 message 事件对其进行捕捉。由服务器发送的数据会保存于 message 事件对象的 data 属性之中。代码清单 17.6 是一个例子。

代码清单 17.6　消息的收发

```
// 向服务器发送消息
ws.send('Hello, WebSocket!');

// 接收服务器发送的消息
ws.onmessage = function(event) {
    // 取出所受到的数据
    var receivedMessage = event.data;

    /* 进行一些处理 */
};
```

在执笔本书时，根据 WebSocket 的标准，只能收发一部分的 JavaScript 对象。不过，由于可以收发任意的字符串，因此可以像代码清单 17.7 这样，通过 JSON.parse 与 JSON.stringify 方法，简单地实现对 JavaScript 对象的处理。

代码清单 17.7　收发任意的 JavaScript 对象

```
var obj = { x:1, y:2 };
```

```
// 将 JavaScript 对象转换为 JSON 字符串并发送
ws.send(JSON.stringify(obj));

ws.onmessage = function(event) {
    // 将接收到的 JSON 字符串转换为 JavaScript 对象
    var receivedObject = JSON.parse(event.data);

    /* 进行一些处理 */
};
```

17.2.3 连接的切断

可以通过在客户端调用 close 方法，来显式地切断连接。一旦连接被切断，就会触发 close 事件，因此需要设定事件处理程序，并实现连接关闭后所必要的处理操作。代码清单 17.8 是一个例子。

代码清单 17.8 连接的切断

```
// 切断连接
ws.close();

ws.onclose = function(event) {
    /* 进行一些处理 */
};
```

不过，在通常的 WebSocket 应用开发中，很少要在客户端切断连接。反而在客户端从长时间的休眠状态中恢复时，或者在 WebSocket 服务器重启时，常常会出现连接被意外切断的情况。

为了应对这种情况，应当在客户端实现能够自动尝试重新连接的功能。这种方式是更为用户友好的。代码清单 17.9 中是一个捕捉了 close 事件，并尝试再次连接的示例。在这一示例中，为了避免产生大量空循环，而使用 setTimeout 来调整重新连接的时间间隔。

代码清单 17.9 再次建立连接

```
//用于保存 WebSocket 实例的变量
var ws = null;

// WebSocket 的初始化
function initWebSocket() {
    ws = new WebSocket('ws://www.foo.org/bar');
    ws.onopen = function(){ /* 进行一些处理 */ };
    ws.onmessage = function(){ /* 进行一些处理 */ };
    // 在连接被切断 10 秒后尝试再次连接
    ws.onclose = function(){ setTimeout(initWebSocket, 10000); };
}
```

17.2.4 连接的状态确认

可以通过引用 WebSocket 实例的 readyState 属性，来确认连接的状态。readyState 所能取得的值，是在 WebSocket 类中被定义的常量属性。表 17.1 总结了 readyState 所能获取的连接状态。

表 17.1 WebSocket 类的常量属性一览

属性	整数值	说明
CONNECTING	0	正在连接
OPEN	1	已处于连接中
CLOSING	2	正在切断连接
CLOSED	3	已经切断或连接失败

17.2.5　二进制数据的收发

WebSocket 所能够收发的格式有字符串、Blob，以及 ArrayBuffer 这三种。Blob 与 ArrayBuffer 是用于通过 JavaScript，来对二进制数据进行处理的格式 ①。如果要在 WebSocket 中发送二进制数据，只需要直接将 Blob 或 ArrayBuffer 直接指定给 send 方法即可。代码清单 17.10 是一个二进制数据的发送方法的示例。

代码清单 17.10　二进制数据的发送

```
// 指定 Blob 并发送
ws.send(blog);

// 指定 ArrayBuffer 并发送
ws.send(arrayBuffer);
```

二进制数据的接收方法与通常的文本数据的接收方法基本一致。为了选择以 Blob 还是 ArrayBuffer 格式来接收，我们需要将 binaryType 属性指定为字符串 "blob" 或 "arraybuffer"。默认情况下的设定值为 "blob"。代码清单 17.11 是一个二进制数据的接收方法的示例。

代码清单 17.11　二进制数据的接收

```
// 指定接收二进制数据的格式
ws.binaryType = 'blob';

ws.onmessage = function(event) {
    var receivedData = event.data;
    if (receivedData.constructor === Blob) {
        // 接收二进制数据

    } else if (receivedData.constructor === String) {
        // 接收文本数据
    }
};
```

17.2.6　WebSocket 实例的属性一览

表 17.2 总结了 WebSocket 类的实例所具有的属性，根据浏览器的种类及版本的不同，有一些属性尚未实现，请酌情通过用于开发的控制台等方式，确认是否进行了实现。

表 17.2　WebSocket 实例的属性一览

属性名	说明
URL	连接目标的 URL
protocol	所选择的子协议名
readyState	连接状态。关于可以通过该属性获取的值，请参见 17.2.4 一节
bufferAmount	在 send 方法中，注册为发送队列的字符串的，尚未发送部分的缓冲大小（单位为字节）
onopen	在连接建立时所执行的事件处理程序
onclose	在连接被切断时所执行的事件处理程序
onerror	在出错时所执行的事件处理程序
onmessage	在从服务器收到消息时所执行的事件处理程序
binaryType	指定了二进制数据的接收格式（'blob' 或 'arrayBuffer'）
send(data)	用于向服务器发送消息的方法（data 可以被指定为 String、Blob 及 ArrayBuffer 中的一种）
close()	用于切断与服务器的连接的方法

① 不过，在执笔本书时，还没有浏览器对二进制数据的收发进行实现。

17.3 WebSocket 实践

接下来，我们将实际地创建一个聊天应用程序，并在此过程中确认 WebSocket 的使用方法。聊天程序是 WebSocket 的范例中最为典型的一种。其处理较为单一，理解起来也比较容易，而且还能够轻松地进行自定义扩展，是一种非常合适的入门主题。

17.3.1 Node.js 的安装

WebSocket 无法像其他 API 那样，仅通过客户端的实现就能确认其运行情况。还必须在服务器端执行大量的功能实现。可以使用任何语言来对服务器端进行实现。大部分主流的语言，都已经发布了用于 WebSocket 服务器实现的库。

在这个教程中，我们将始终 Node.js[①]，而在服务器端则会使用 JavaScript 来进行实现。Node.js 运行于开源的高速 JavaScript 引擎 V8 之上，是一种目前发展势头良好的服务器端 JavaScript 实现。本书第 6 部分将会对其进行详细说明，因此现在只简单地作为参考即可。

首先需要安装 Node.js。Node.js 现在仍然处于活跃开发状态，会频繁地进行版本升级。因此，已经发布了多个用于管理 Node.js 主体版本的工具。在这里使用的 Node.js 版本管理工具是 nave，接下来，我们来介绍使用 nave 进行安装的方法（图 17.2）。

图 17.2 Node.js 主体的安装

```
% git clone git://github.com/isaacs/nave.git ~/.nave
% ~/.nave/nave.sh ls-remote
remote:
0.0.1    0.0.2    0.0.3    0.0.4    0.0.5    0.0.6    0.1.0    0.1.1    0.1.2
0.1.3    0.1.4    0.1.5    0.1.6    0.1.7    0.1.8    0.1.9    0.1.10   0.1.11
0.1.12   0.1.13   0.1.14   0.1.15   0.1.16   0.1.17   0.1.18   0.1.19   0.1.20
0.1.21   0.1.22   0.1.23   0.1.24   0.1.25   0.1.26   0.1.27   0.1.28   0.1.29
0.1.30   0.1.31   0.1.32   0.1.33   0.1.90   0.1.91   0.1.92   0.1.93   0.1.94
0.1.95   0.1.96   0.1.97   0.1.98   0.1.99   0.1.100  0.1.101  0.1.102  0.1.103
0.1.104  0.2.0    0.2.1    0.2.2    0.2.3    0.2.4    0.2.5    0.2.6    0.3.0
0.3.1    0.3.2    0.3.3    0.3.4    0.3.5    0.3.6    0.3.7    0.3.8    0.4.0
0.4.1    0.4.2    0.4.3    0.4.4    0.4.5    0.4.6    0.4.7    0.4.8

% ~/.nave/nave.sh install 0.4.8
% ~/.nave/nave.sh use 0.4.8
% echo "~/.nave/nave.sh use 0.4.8">> ~/.bashrc
```

在安装了 Node.js 的主体之后，接下来要安装的是 npm（图 17.3）。npm 是 Node.js 使用的包管理器。通过 npm，就能很容易地安装适用于 Node.js 的库。这次将会安装的是 websocket-server，它是一种用于实现 WebSocket 服务器的库。应当在开发文件夹内执行 npm install 指令，这是因为该库将会被部署于执行了 npm 指令的文件夹中。

图 17.3 npm 与 websocket-server 的安装

```
% curl http://npmjs.org/install.sh | sh
% npm install websocket-server
websocket-server@1.4.04 ./node_modules/websocket-server
```

如果是在 Windows 中尝试安装，请使用 Cygwin 或通过 VirtualBox 使用 Ubuntu 等系统来完成操作。而如果 Node.js 是 0.5.1 或更新版本，则可以在 Windows 中使用 "node.exe" 来方便地执行 Node.js 程序。其官方网站已经发布了这一工具，不过在使用 node.exe 时，npm 之类的工具还是无法使用，需要以手动

① http://nodejs.org

的方式来安装包。

17.3.2　服务器端的实现

在开始进行实现之前，我们先要对一些常见的 WebSocket 应用程序的模式进行说明。WebSocket 服务器在收到消息之后，将会进行一些处理，并将消息返回至连接目标。这时，通常的 WebSocket 应用程序的返回目标，大致可以分为以下 3 种模式。

- 将数据返回给消息的发送者
- 向所有人广播该消息
- 仅将消息广播给符合条件的人

对于聊天程序来说，只要向所有处于连接状态的人广播消息，就能够实现最低程度的功能。如果还希望在此增加聊天室等功能，则需要对相应的广播处理进行实现，将发言内容的可见性限制在聊天室内的人之间。

这里实现的 WebSocket 服务器，将会把所接收到的消息广播至所有处于连接中的人。这实在是一种很常见的功能，恐怕在大多数的库中，都已经准备了能够实现这一功能的方法了。Node.js 的 websocket-server 自然也准备了相应的方法。代码清单 17.12 是一个实现示例，请以任意的名称，将其保存于安装了 websocket-server 的文件夹中。这里使用的是 chat.js。

代码清单 17.12　WebSocket 服务器的实现示例（chat.js）

```
// 使用 websocket-server
var ws = require('websocket-server');

// 创建 WebSocket 服务器
var server = ws.createServer();

// 捕捉连接事件
server.addListener('connection', function(socket) {
    // 显示日志
    console.log('onconnecion:', socket);

    // 捕捉消息的接收事件
    socket.addListener('message', function(data) {
        // 将所接收的消息直接广播给所有处于连接中的人
        server.broadcast(data);
    });
});

// 在 8888 号端口处理访问
server.listen(8888);
console.log('waiting...');
```

几乎上面的每一行都添加了注释，不过其实这是一段只有 10 行左右的代码。通过它就能够实现一个简单的 WebSocket 服务器了。为了启动服务器，还需要执行以下指令。至此，服务器端的实现就全部完成了。

```
% node chat.js
waiting...
```

17.3.3　客户端的实现

代码清单 17.13 是一个客户端的实现示例。这里所实现的处理大致上可以分为，将文本框中的字符串发送给服务器的处理，以及显示从服务器收到的数据的处理。可以在 WebSocket 构造函数中指定 WebSocket 服务器的 URL。如果是在本地运行 Node.js，则会使用本例中这样的 URL。

还需要以任意的名称保存文件，并将其配置为 WebSocket 服务器所能引用的路径。这里所使用的是 chat.html。说一句题外话，在 HTML5 中，html 标签及 body 标签可以省略。因此，即使是代码清单 17.13 中的代码，也能够被作为 HTML5 标记语言来正确地进行解释。

代码清单 17.13 聊天客户端的实现示例（chat.html）

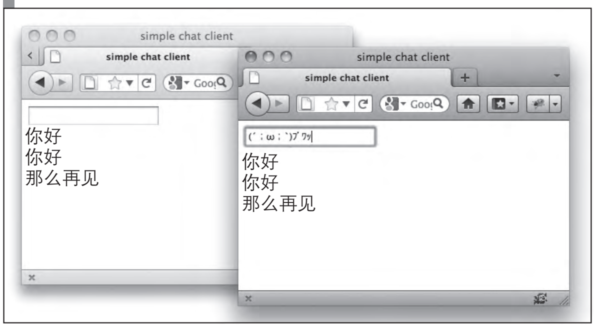
```
<!DOCTYPE html>
<title>simple chat client</title>
<!-- 通过回车键发送文本 -->
<input onkeydown="if(event.keyCode===13)submit(this)">
<script>
var ws = new WebSocket('ws://localhost:8888/');

// 将接收的消息插入 body 内
ws.onmessage = function(event) {
    var comment = document.createElement('li');
    comment.textContent = event.data;
    document.body.appendChild(comment);
};

// 将文本框的值发送给服务器
function submit(textbox) {
    ws.send(textbox.value);
    textbox.value = '';
}
</script>
```

之后请同时在多个标签页中打开 chat.html。如果连接成功，启动了 WebSocket 服务器的终端应该会输出日志。在文本框中输入消息并按下回车键之后，消息将通过 WebSocket 服务器广播，所有打开着的页面都将会显示该消息（图 17.4）。

图 17.4 简单的聊天应用程序



17.3.4 客户端的实现 2

尽管本章将上面的程序称为一个范例，不过它并没有什么实际意义。因此，接下来将会尝试在聊天中添加用户名，以及能够自由设定评论样式的功能。此外，还会试着添加一些功能，使聊天对象的输入状态得以可视化。

这次修改不会对服务器端的代码作任何更改。接下来，将会对之前的范例分别添加两条逻辑，其一是用于在客户端构造所要发送的 JSON 的逻辑，另一个则是用于绘制所广播的 JSON 的逻辑（代码清单 17.14）。

代码清单 17.14 聊天客户端的实现示例（chat2.html）

```
<!DOCTYPE html>
<title>simple chat client</title>
<input id="user">
<input id="css">
<input id="text">
<div id="typing"></div>

<script>
```

```
var ws = new WebSocket('ws://localhost:8888/'),
        $ = function(id){ return document.getElementById(id); };
$('text').onkeydown = function(event) {
    // 通过回车键以向服务器发送发言
    if (event.keyCode === 13) {
        ws.send(JSON.stringify({
            action: 'post',
            user: $('user').value,
            css: $('css').value,
            text: $('text').value
        }));
        $('text').value = '';

    // 向服务器发送正在输入这一状态
    } else {
        ws.send(JSON.stringify({ action:'typing', user:$('user').value }));
    }
}

ws.onmessage = function(event) {
    var data = JSON.parse(event.data);

    // 根据 action 属性来切换相应的处理
    switch(data.action) {
        case 'post':      // 绘制发言
            var comment = document.createElement('li');
            comment.style.cssText = data.css;
            comment.textContent = data.user + ': ' + data.text;
            document.body.appendChild(comment);
            break;

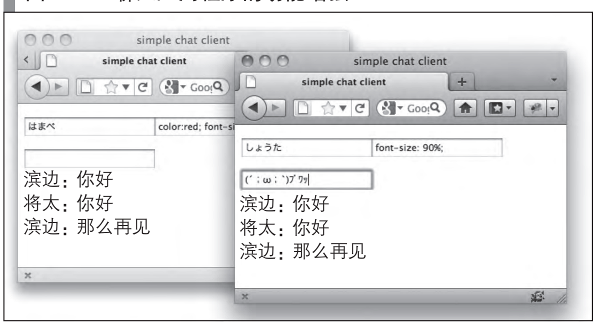
        case 'typing':         // 绘制正在输入的状态
            $('typing').textContent = data.user + ' 先生 / 小姐正在输入……';
            clearTimeout(timer);
            timer = setTimeout(function(){ $('typing').textContent = ''; }, 3000);
            break;
    }
};
</script>
```

虽然增加的功能很简单，但在增加功能的过程中，的确没有在服务器端新增任何的逻辑（图 17.5）。在 WebSocket 应用程序中，只要有一个这样的具备有广播功能的服务器，并且在客户端实现了 JSON 构造及绘制的逻辑的话，就能够只通过客户端的代码来实现各种各样的创意。

例如，只要在广播的消息中添加信息并将其返回，就连类似于“赞”按钮或问卷之类的功能也能被实现。而这仅需要修改客户端即可。特别是对于这样的实时消息收发来说，在服务器端连数据库都不必准备。

当然，在实际发布服务时，在服务器端也有很多应当考虑之处。不过作为 WebSocket 应用程序开发的练习课题来说，这样的实时消息收发程序已经足够了，强烈推荐大家试一下。

图 17.5　聊天应用程序的功能增强



第 18 章 Web Workers

本章将会说明 Web Workers。单线程是 JavaScript 中习以为常的规范了，而 Web Workers 则将多线程的概念引入了 JavaScript。随着 Web Workers 的引入，很多在此之前被认为难以在客户端实现的问题，将可能得以解决。

18.1 Web Workers 概要

18.1.1 Web Workers 的定义

从内部来看，客户端 JavaScript 与 UI 的渲染处理用的是同一个进程，而且是以单线程的方式执行的。因此，如果在客户端 JavaScript 中执行高负载的处理，就可能会发生 UI 渲染处理被阻断这一致命问题。

Web Workers[1] 是一种能够在另外的线程中创建新的 JavaScript 运行环境，以使 JavaScript 代码能够在后台处理的一种机制。如果能够视情况恰当地分离出复杂的处理，并将其置于后台运行，就能够在通过客户端进行复杂处理的同时，不妨碍用户的 UI 操作，从而开发出高可用性的 Web 应用程序。

此外，本书已经介绍过的 File API（15.2 节）和 Indexed Database（16.2 节）等 I/O 处理 API 中，都已经提供了不会阻断 UI 的事件驱动 API。不过，Web Workers 的运行环境是与 UI 处理线程相分离的，而不用担心将会阻断 UI，因此，它们同时也提供了一些仅能够运行于 Web Workers 环境的简单的同步 I/O 处理 API。关于同步 API，在各章都有所简单介绍，请分别参考相关章节。

18.1.2 Web Workers 的执行方式

首先，我们对一些术语进行定义。本书将通常的客户端 JavaScript 运行环境称作主线程，而将通过 Web Workers 创建的后台 JavaScript 运行环境称为工作线程（Worker）。可以在主线程中创建工作线程，且能够同时创建多个工作线程。

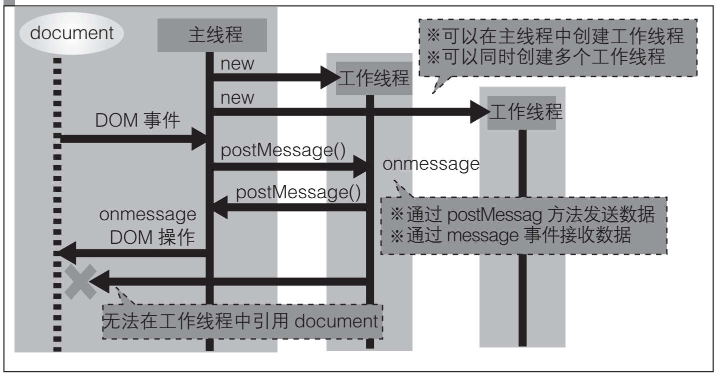
主线程与工作线程的 JavaScript 运行环境是相互分离的，无法相互引用对方环境中的变量。也就是说，在各自的环境中分别准备了其全局对象，且这些全局对象无法被相互引用。可以通过 window 这一名称来引用主线程的全局对象，并通过 self 这一名称来引用工作线程的全局对象。

还有一个重要的问题是，从工作线程的环境中是无法引用 document 对象的。也就是说，所有的 UI 操作，即 DOM 的引用与更改，都只能在主线程中进行。无法在工作线程中对 UI 进行任何操作。如果要进行数据的收发处理，则必须通过消息收发接口（postMessage 方法、message 事件）来进行。

这些都是为了简化多线程程序设计而添加的限制。在语言规范层面上保证了 UI 操作始终都只能在主线程中执行的话，就能够将一些在多线程程序设计中多发的错误防范于未然。图 18.1 展示了主线程与工作线程的关系。

[1] http://dev.w3.org/html5/workers

图 18.1　主线程、工作线程与 DOM 的关系图



18.2 基本操作

18.2.1 工作线程的创建

工作线程的创建是非常简单的。可以通过在主线程中调用 Worker 构造函数来创建工作线程。

```
var worker = new Worker('worker.js');
```

对于希望在工作线程中执行的 JavaScript 代码，只要将写有该代码的文件的 URL 指定为构造函数的参数，就能够使其在后台运行。在上面的示例中，当前文件夹中的 worker.js 将被下载，而写于 worker.js 中的代码则会在后台执行。

对于在 Worker 的构造函数中所指定的文件，也有同源策略的限制。因此，主线程只能够读取与其同源的文件，对此请加以注意。

18.2.2 主线程一侧的消息收发

可以通过调用 Worker 实例的 postMessage 方法，来从主线程向工作线程发送消息。postMessage 所能发送的数据类型是很自由的，任意的 JavaScript 对象都能够被发送。不过，包括 document 对象等一些特殊的对象，是无法被发送的。

可以通过捕捉 Worker 实例的 message 事件，以接收来自工作线程的消息。而工作线程实际发送的数据本身，是被保存于 message 事件对象的 data 属性之中的。代码清单 18.1 是一个例子。

代码清单 18.1　主线程一侧的消息收发

```
// 消息的发送
worker.postMessage('foo');
worker.postMessage(100);
worker.postMessage({ x:1, y:2 });

// 消息的接收
worker.onmessage = function(event) {
    // 获取所发送的数据
    var receivedMessage = event.data;

    /* 进行一些处理 */
};
```

18.2.3 工作线程一侧的消息收发

我们可以通过调用在工作线程中所定义的 postMessage 方法，来从工作线程向主线程发送消息。此时，和之前一样，任意的 JavaScript 对象都能够被发送。

我们可以通过捕捉全局对象的 message 事件，来接收发送自主线程的消息。而主线程实际发送的数据本身，是被保存于 message 事件对象的 data 属性之中的。代码清单 18.2 是一个例子。

代码清单 18.2 工作线程一侧的消息收发

```
// 消息的发送
self.postMessage('foo');
self.postMessage(100);
self.postMessage({ x:1, y:2 });

// 消息的接收
self.onmessage = function(event) {
    // 获取所发送的数据
    var receivedMessage = event.data;

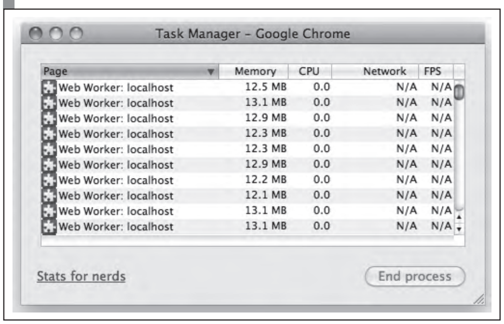
    /* 进行一些处理 */
};
```

此时消息收发的接口与在主线程中的情况几乎相同。在这里，工作线程内的 self 都可以省略，但为了更容易地区分调用方是主线程还是工作线程，本书并不会省略它们，而是会一一写明。

18.2.4 工作线程的删除

即使是小型的程序，工作线程也会占用一定程度的内存，因此，最好删除不必要的工作线程。对于 Chrome 来说，可以在选项菜单中的工具列表中启动任务管理器，通过它来确定工作线程的内存及 CPU 占用状态（图 18.2）。

图 18.2 Chrome 的任务管理器状况显示



有两种方法可以用来删除工作线程，分别是在主线程进行删除的方法和在工作线程中对自身进行删除的方法。如果要在主线程删除一个工作线程，我们可以调用 Worker 实例的 terminate 方法。如果要在工作线程中删除自身，则可以调用 close 方法（代码清单 18.3）。

代码清单 18.3 工作线程的删除

```
// 在主线程中进行删除的情况
worker.terminate();

// 在工作线程中删除自身的情况
self.close();
```

不过，要再次创建工作线程的成本也并不低，所以，如果要频繁地调用工作线程进行处理，或创建工作线程时需要花费相当的时间时，则不建议每次都删除工作线程，而应该反复使用已有的工作线程。

18.2.5　外部文件的读取

可以通过调用 importScripts 方法，以在工作线程内部读取外部的 JavaScript 文件。importScript 可以读取非同源的文件，所以在有些情况下，能够实现比 JSONP 方式更为便捷易用的跨源通信。importScript 是采用同步的方式读取文件的，所以不必考虑文件的读取等待问题。其参数可以同时指定多个文件（代码清单 18.4）。

代码清单 18.4　importScript 的使用示例

```
// 外部 JavaScript 文件的读取
self.importScripts('http://www.foo.org/external.js', 'dependent.js');

// 此时，外部 JavaScipt 文件中的内容已经被分析求值
dependent.some_method('foo', 'bar');
```

18.3　Web Worker 实践

接下来，请大家尝试一下使用工作线程来进行程序设计。在接下来将要编写的范例中，在文本框中输入了文字之后，就会在客户端搜索用户名，并将其显示出来。由于搜索操作是通过 Web Workers 在后台进行的，因此即使正在进行复杂的搜索处理时，也不会阻断 UI 刷新。本节将会在实现具有实际意义的搜索功能的同时，对 Web Workers 程序设计的要点进行说明。

18.3.1　工作线程的使用

首先，我们通过使用工作线程，来实现一个具有简单的搜索功能的范例。代码清单 18.5 中是工作线程中的源代码。

代码清单 18.5　使用工作线程实现搜索功能的示例

```
// 用户数据
var userList = [
    'Seiichiro INOUE',
    'Shota HAMABE',
    'Takuro TUCHIE',
    /* 及更多用户 */
}

// 接收消息
self.onmessage = function(event) {

    // 通过所接收的消息来创建正则表达式
    var reg = new RegExp(event.data, 'i'),
        html = '';

    // 搜索与正则表达式相匹配的用户
    userList.forEach(function(user) {
        if (reg.test(user)) {
            html += '<li>' + user + '</li>';
        }
}

    // 发送包含搜索结果的 HTML 字符串
    self.postMessage(html)
};
```

在工作线程中，将根据所接收到的字符串来创建正则表达式，并以该正则表达式按顺序搜索大量用户数据，在对所有的用户数据完成检查之后，将匹配结果返回给主线程。

这里的要点在于，由于应尽可能地在工作线程中，对所有工作线程能够进行的操作进行处理，因此 HTML 字符串的构造也是在工作线程中完成的。而随着主线程所要执行的任务的减轻，使得在执行搜索功能时，其 UI 操作更不易发生阻塞。

此外还有另一种实现方法，即通过数组的形式将搜索结果返回至主线程，然后在主线程中对 HTML 进行构造。这种方法的优点在于，与之前的做法相比，设计与逻辑部分被很好地分离开了。

这一范例将更重视搜索的速度，所以将会选择前一种方法。

代码清单 18.6 中是主线程中的源代码。

代码清单 18.6　主线程中的实现示例

```
<!DOCTYPE html>
<input type="text" id="textbox">
<div id="results"></div>
<script>
    // 创建工作线程
    var worker = new Worker('worker.js'),
        textbox = document.getElementById('textbox'),
        results = document.getElementById('results');

    // 接收来自工作线程的搜索结果
    worker.onmessage = function(event) {
        results.innerHTML = event.data;
    };

    // 将文本框中的内容发送至工作线程
    textbox.onkeyup = function(event) {
        worker.postMessage(textbox.value);
        results.innerHTML = '';
    };
</script>
```

在主线程中，将会在文本框中有按键输入时向工作线程发送搜索请求。由于使用了工作线程，因此无论用户数据增加多少，搜索处理都不会阻塞 UI 操作（不过，如果要一次性绘制大量的搜索结果，则仍有可能会使 UI 操作阻塞）。如果没有使用工作线程，则会在搜索用户数据的过程中阻塞 UI 的更新，连文本框的输入操作都将无法进行。

18.3.2　中断对工作线程的处理

其实在之前的范例（代码清单 18.5、代码清单 18.6）中，有一个很严重的问题。这个问题就是，如果工作线程在执行搜索处理的过程中，又收到了下一个搜索请求的话，则会等待之前的搜索处理结束，再开始新的处理。

为了解决这一问题，需要使工作线程的处理能够在任意时刻被中断。对此，主要有两种方法。

- **重新创建工作线程（在主线程中解决这一问题）**
- **使用计时器（在工作线程中解决这一问题）**

■ 重新创建工作线程

在第一种方法中，主线程将会在发送消息时重新创建工作线程。只要丢弃该工作线程，其正在执行的处理也就会被强制中断。代码清单 18.7 是一个示例。

这种方法的好处在于，无需修改工作线程中的代码。而相对地，其缺点在于产生了重新创建工作线程的成本。对于本例中这样需要频繁调用工作线程的情况来说，创建工作线程所花费的成本是不容小视的。

代码清单 18.7　通过重新创建工作线程来实现处理的中断

```
var worker = null;

// 在输入后，重新创建工作线程
textbox.onkeyup = function(event) {
    // 强制结束已有的工作线程
    if (worker) {
        worker.terminate();
        worker = null;
    }

    // 工作线程的创建
    worker = new Worker('worker.js');
    worker.onmessage = function(event) {
        results.innerHTML = event.data;
    }

    // 将文本框中的内容发送至工作线程
    worker.postMessage(textbox.value);
    results.innerHTML = '';
};
```

■ 使用计时器

另一种方法是在工作线程中通过计时器中断处理。如果将复杂的处理分割成小块，并通过计时器来执行，就能够在处理的切分处插入 message 事件，并通过取消计时器来实现处理的中断操作。代码清单 18.8 是一个例子。

这种方法的好处在于，在调用时不会产生创建工作线程的开销。其缺点则是由于要通过计时器实现分割处理，因此工作线程的代码复杂度将会增加，从而使得完成整个处理所需的总时间增长。

在现在的例子中，我们将会以 100 行为单位切分用户数据，以执行搜索处理。在所有的搜索处理都完成之后，将会返回搜索结果。在实际中，以多大的单位分割都可以，同时也并不需要将所有的结果一起返回。对于各种不同的情况，有着不同的最佳分割方案，应有针对性地选择合适的方法。

代码清单 18.8　通过计时器来实现处理的中断

```
// 用户数据
var userList = [
    'Seiichiro INOUE',
    'Shota HAMABE',
    'Takuro TUCHIE',
    /* 及更多用户 */
};

var timer = null;
self.onmessage = function(event) {
    // 中断正在执行的处理
    clearInterval(timer);

    // 将值重置
    var reg = new RegExp(event.data, 'i'),
        html = '',
        pos = 0;

    // 通过计时器进行分割并执行
    timer = setInterval(function() {
        // 以 100 行为单位进行处理
        userList.slice(pos, (pos += 100)).forEach(function(user) {
            if (reg.test(user)) {
                html += '<li>' + user + '</li>';             }
        });

        // 如果完成了所有的处理，则返回搜索结果
```

```
        if (pos >= userList.length) {
            self.postMessage(html);
            clearInterval(timer);
        }
    }, 0);
};
```

18.4 共享工作线程

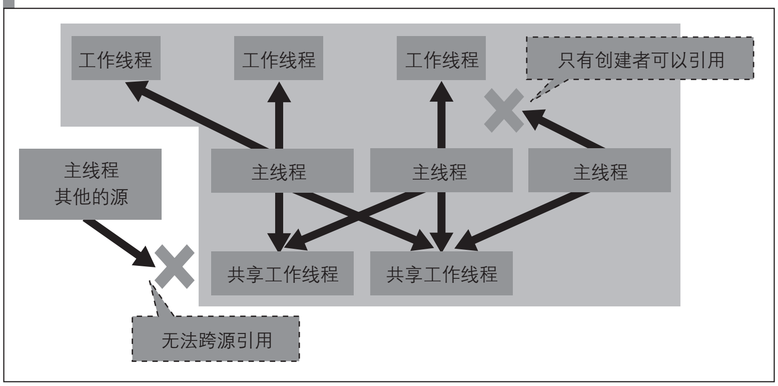
18.4.1 共享工作线程的定义

之前所讲的工作线程，总是与某一特定对象具有一一对应的关系。不过，在 Web Workers 中，也提供了工作线程的共享功能，这种情况下，1 个工作线程可以被多个页面共享引用。为了区分于通常的工作线程，这种类型的工作线程被称为“共享工作线程”（图 18.3）。此时同源策略的限制仍然存在，不过已经能够同时在多个不同的窗口之间引用同一个共享工作线程。

在使用了共享工作线程之后，需要创建的工作线程的数量将会减少，从而节约了资源。同时在创建了工作线程之后就不需要再次重复创建，也就避免了工作线程的创建开销。

共享工作线程有多种应用方式，举例来说，可以通过共享工作线程来实现窗口间的消息收发，或者以共享工作线程作为中转，将服务器连接相整合。这些技术使得各种各样的应用方式成为可能。

图 18.3 工作线程与共享工作线程



18.4.2 共享工作线程的创建

我们可以通过调用 SharedWorker 类的构造函数来创建共享工作线程。其第 1 个参数和通常的工作线程一样，需要指定一个 JavaScript 文件的 URL，而其第 2 个参数所指定的则是该共享工作线程的名称。如果省略了第 2 个参数，则会默认使用空字符。

此时，如果在不同的窗口中，以同样的 JavaScript 文件及共享工作线程名来执行 SharedWorker 构造函数，且该共享工作线程已经被创建的话，则会返回一个引用了相同共享工作线程的 SharedWorker 实例。代码清单 18.9 是一个例子。其中的 worker1 与 worker2 引用了同一个共享工作线程。

代码清单 18.9 共享工作线程的创建

```
// 创建共享工作线程（在 http://example.com/foo.html 中执行）
var worker1 = new SharedWorker('worker.js', 'test-worker');
```

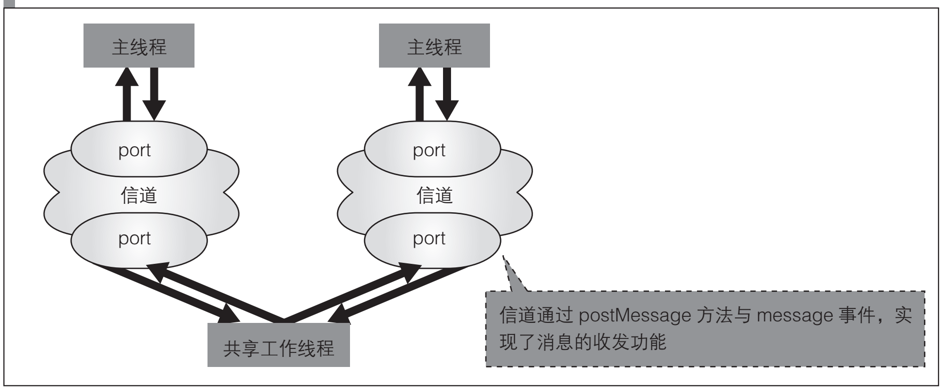
```
// 创建共享工作线程（在 http://example.com/bar.html 中执行）
var worker2 = new SharedWorker('worker.js', 'test-worker');
```

18.4.3　共享工作线程的消息收发

对于通常的工作线程来说，消息的收发是通过全局对象的的 postMessage 方法与 message 事件来实现的。而对于共享工作线程来说，能够进行消息收发的目标不止一个，所以需要有一些机制来实现与特定目标之间的消息收发。

在使用共享工作线程进行消息的收发时，其内部使用了信道通讯[①]。所谓信道通讯指的是，一种通过名为 MessagePort 的成组对象进行消息收发的机制。在一个 MessagePort 对象中调用 postMessage 的话，另一个 MessagePort 对象的 message 事件就会被触发（图 18.4）。

图 18.4　MessagePort 的概念图



在主线程中创建的 SharedWorker 实例具有 port 这一属性名，该属性可以引用 MessagePort 对象的其中一个。我们可以在主线程中通过该 port，来实现与通常的工作线程相同的消息收发操作。代码清单 18.10 是一个例子。

代码清单 18.10　在主线程中进行消息收发

```
// 共享工作线程的创建
var worker = new SharedWorker('http://www.foo.org/bar.js');

// 通过 MessagePort 发送消息
worker.port.postMessage('foo');
// 通过 MessagePort 接收消息
worker.port.onmessage = function(event) {
    var receivedData = event.data;
    /* 进行一些处理 */
};
```

在主线程中创建了 SharedWorker 实例之后，就会触发共享工作线程的 connect 事件。这时，在 connect 事件对象中有一个 MessagePort 对象，该对象与请求连接的主线程中所具有的 MessagePort 对象是相对应的。通过这一 MessagePort 来发送与接受消息的话，就能够与请求连接的主线程进行消息收发处理。代码清单 18.11 是一个例子。

代码清单 18.11　在共享工作线程中进行消息收发

```
// 来自于主线程的连接请求
```

① http://dev.w3.org/html5/postmsg/#channel-messaging/

```
self.onconnect = function(connectEvent) {
    // 获取连接请求方的 MessagePort
    var port = connectEvent.ports[0];

    // 通过 MessagePort 发送欢迎信息
    port.postMessage('Hello, SharedWorker!');

    // 通过 MessagePort 接收消息
    port.onmessage = function(messageEvent) {
        // 将所接收的数据直接回复
        port.postMessage(messageEvent.data);
    };
};
```

18.4.4 共享工作线程的删除

由于可以同时在多个窗口中引用共享工作线程，因此与通常的工作线程不同，SharedWorker 实例中是没有 terminate 方法的。如果希望在主线程中关闭某些连接，需要调用 MessagePort 的 close 方法。

```
worker.port.close();
```

而即使在主线程中调用了 close 方法，只要还有其他窗口正在对共享工作线程进行引用，该线程就不会被删除。只有在最后一个引用也被释放时，该共享工作线程才会被删除。不过，和通常的工作线程一样，随时都能通过调用共享工作线程中的 close 方法来关闭自身。

```
self.close();
```

18.4.5 共享工作线程的应用实例

■ 窗口间通信

通过对共享工作线程的 MessagePort 进行管理，就能够借助该共享工作线程实现在不同窗口之间的消息收发。在代码清单 18.12 的例子中，共享工作线程所接收的消息，将会被广播至所有处于连接状态的窗口。

代码清单 18.12 通过共享工作线程实现的窗口间通信

```
var ports = [];

self.onconnect = function(connectEvent) {
    // 获取连接请求方的 MessagePort
    var port = connectEvent.ports[0];

    // 在数组中保存所有的 MessagePort
    ports.push(port);

    // 向所有的窗口发送消息
    port.onmessage = function(messageEvent) {
        ports.forEach(function(e) {
            e.postMessage(messageEvent.data);
        });
    };
};
```

如果只希望像代码清单 18.12 那样，向所有的窗口发送消息，只需要将所有的 MessagePort 保存于数组中即可实现。如果希望将消息发送给某一指定的窗口，则需要将用于确定特定窗口的名称与 MessagePort 相关联并进行管理。

■ 服务器连接的整合

共享工作线程还有一种应用方式，即可以将共享工作线程作为中转，将服务器连接结合为一个整体。试考虑一个通过 WebSocket 与服务器进行通信的应用程序。如果仅仅是复制了窗口，则会建立和窗口数量相同的 WebSocket 连接，从而造成网络及 CPU 资源的浪费。

对于这种情况，如果通过 1 个共享工作线程来对 WebSocket 通信进行处理的话，即使同时打开了多个窗口，也能将共享工作线程作为中转来整合这些 WebSocket 连接。对其进行了引用的窗口发送的消息将会被 WebSocket 服务器所接收，而服务器所发送的消息则会被分发至所有对其进行了引用的窗口。代码清单 18.13 是其示例。

代码清单 18.13　WebSocket 连接的整合

```
// 建立 WebSocket 连接
var ws = new WebSocket('ws://www.foo.org/chat');

var posts = [];

self.onconnect = function(connectEvent) {
    // 将所有的 MessagePort 保存至数组之中
    var port = connectEvent.ports[0];
    ports.push(port);

    // 将窗口发出的消息传递至 WebSocket 服务器
    port.onmessage = function(messageEvent) {
        ws.send(messageEvent.data);
    };
};

// 将来自于 WebSocket 服务器的消息发送至所有的窗口
ws.onmessage = function(event) {
    ports.forEach(function(port) {
        port.postMessage(event.data);
    });
};
```



第 5 部分

Web API

在服务器端对 Web API 进行调用是当前的主流方式。如果在客户端调用 Web API，则会有跨源限制及权限授予等问题。不过，现在也出现了回避这些问题，并通过客户端 JavaScript 来调用 Web API 的做法。

第 19 章 Web API 的基础

本章将边回顾 Web API 的历史边说明 Web API。在通过客户端 JavaScript 调用 Web API 时，会产生跨源限制的问题。本章也将对该问题的回避方法进行说明，最后还会介绍 OAuth 的相关内容，在今后的 Web API 中，这将是一种很重要的权限授予协议。

19.1 Web API 与 Web 服务

API 这一术语指的是应用程序与系统之间的接口，是 Application Programming Interface 的缩写。在 Web API 这一术语出现之前，API 主要的目标使用对象是操作系统及框架。在特定的操作系统（Unix 或 Windows 等）或特定的框架上运行的程序，将会通过操作系统及框架所提供的功能来执行处理。

从程序开发者的角度来看，程序代码所使用的是库（Unix 系操作系统中的 libc 或 Windows 中的 DLL 等）中的函数或类库。也就是说，在开发者看来，API 是函数以及类的标准，是由他们所决定的。

Web API 的目标使用对象

■ Web API 的定义

Web API 的目标使用对象是 Web 服务。本节之后将说明 Web 服务这一术语的定义。这里，我们首先假设存在 Web 服务这一体系，并以此为前提对 Web API 进行定义。

假设存在一些使用了 Web 服务的程序。在 Web 服务与这一程序之间，是通过 HTTP 进行通信的。程序向 Web 服务发出 HTTP 请求，并接收其响应。Web 服务可以提供多种功能，这里假设它可以创建一个文档，并返回处理结果。这时，通信所使用的规则就是 Web API。

接下来将会讲到，Web API 也能具有一些其他的形式。例如，Web API 还可以是在内部隐藏了 HTTP 通信的函数或类库的形式。不过，Web API 最为基本的形式就是一种 HTTP 的调用规则。也就是说，它包含了请求方的 URL 定义、查询参数名、响应格式的定义等内容。

■ Web 服务与 Web 应用程序

本节将继续对 Web 服务这一术语进行说明。在很多情况下，Web 服务与 Web 应用程序的用法没有太大差异，它们之间的区别并不那么明显。两者的基本功能没有区别，都是接收 HTTP 请求并返回响应。有的人将程序所调用的体系称为 Web 服务，而将用户通过浏览器所使用的体系称为 Web 应用程序。

不过，被调用方并没有办法准确的区分自己是被用户使用的，还是被程序使用的，因此，这种定义方式从技术上来说是含糊不清的。本书把从形式上规定（定义）了 HTTP 请求及响应的体系称为 Web 服务，并把没有进行如此规定的体系称为 Web 应用程序。而 Web API 正是一种形式上的规则，因此，Web API 与 Web 服务其实是表里一致的[1]。

① 由于一些历史原因，也有人将 Web 服务等同于 SOAP。这种定义过于狭义，因此，本书没有采用。

19.2 Web API 的历史

19.2.1 Web 抓取

姑且不谈 Web 的最初目的，早期的 Web 页面主要都是一些供用户阅读的文档。不过，文档与数据的区别仅仅是由阅读者决定的。必然会有人希望将在 Web 上公开的文档作为数据处理。Web 文档的标准格式是 HTML，而在将 HTML 文档作为数据处理时，通常会存在两个问题。

首先，HTML 是一种宽松的格式。即使 HTML 中存在一些语法错误，浏览器也能正常显示。因此，一些包含了语法错误的 HTML 也被传播了出去。

而另一个问题则是，在 HTML 中不仅包含了文档的内容，还写有布局及字符修饰等的设计信息。虽然之后出现的 CSS 将这些修饰部分分离到了 CSS 中，但在此之前，HTML 的修饰都是通过 HTML 标签来实现的。而且，在 CSS 出现之后，也不是所有的 HTML 都将设计信息完全地分离到 CSS 中了。从数据处理的角度来看，这些用于控制页面设计的部分，是不需要的多余信息。

在从 HTML 中抽出所需的数据的过程中存在很多问题。HTML 可能具有语法错误，且在抽取过程中，必须去除不需要的信息。通常，这类处理被称为 Web 抓取。

■ Web 抓取所存在的问题

通过正则表达式来抓取的话，代码通常会很复杂。不但难以进行设计，且设计出的代码一般只能用于特定的 Web 页面。也就是说，如果设计了一个用于抽取 Yahoo! 这一网站中的特定页面的程序，它是无法用于其他站点的数据抽取处理的。这是因为，根据页面的不同，HTML 的结构及标签的使用方式也会有所差异。

此外，如果 Web 站点的 HTML 进行了更新，Web 抓取的功能可能就会失效。这是由于 Web 站点的开发者在更新页面时并没有考虑 Web 抓取程序的情况。

19.2.2 语义网

在这样的背景下出现了一种运动，主张将 Web 中的文档从仅供用户阅读的内容，转换为程序也能够进行处理的数据。这一运动的影响范围非常深远，带动了很多相关基本技术的出现。XML 与 CSS 都是在这一连串运动中出现的基本技术。

最能形象地表现这种运动的词是语义网。它还有另一个引人注目之处，即它是由 Web 的创始人，蒂姆・伯纳斯–李（Tim Berners-Lee）所主导的[①]。

有一些概念受到了语义网的直接或间接的影响，而有一些概念虽然与语义网没什么关系，但却是由类似的理念所引出的。接下来，本章将从数据格式与通信协议这两个角度整理说明。

19.2.3 XML

对于 XML 这种常见的数据格式，无需过多说明。XML 也可以被称为元格式。正如其名称所示（XML 的 X 代表的是 eXtensible，意指可扩展的），基于 XML，可以构造出各种各样的数据格式。向下兼容 XML 标准的程序被称为 XML 应用程序。这一术语不太直观，本书将使用 XML 兼容标准的称法。

① 不过很可惜，对于普通用户来说，语义网谈不上是一种成功。不过其理念确实为现在的 Web API 所继承。对于语义网，有着各种不同的见解与立场，因此，这里不再对其作过多的说明。

■ XML 的标准

XML 是通过标签及属性等规则来确定标准的。诸如 <p> 这样的标签，在 HTML 中很常见。在这类兼容标准中，将会把 p 解释为段落（paragraph）的 p。包括 XHTML 在内，存在大量基于 XML 的兼容标准。

对基于 XML 的格式而言，又可以分为仅在形式上遵循 XML 标准的格式，以及作为其兼容标准，对语言标准进行了严密设计的格式这两种类型。

对于前者，举例来说，在希望表示价格时，可以使用名为 cost 的标签，以 <cost>100</cost> 的形式表达。由于这种方式从形式上遵循了 XML 标准，因此可以通过 XML 分析器抽取其中的标签名及元素。不过，因为标签是被随意选取的，所以必须确保数据的创建方与解释方对其有相同的定义。这样的仅在形式上符合标准的 XML 被称为形式良好的 XML 文档。

另一方面，严密设计了 cost 标签的含义的兼容标准，则被称为合法的 XML 文档。

■ XML 的结构描述（Schema）

标签的含义，以及 XML 的构造规则，被称为 XML 的结构描述。用于书写结构描述的具有代表性的语言有 DTD、XML Schema、RELAX NG 等。

DTD 是最古老的描述语言，且没有类型的概念。即对于 cost 标签来说，无法对元素的值是一个数值这一含义进行表述。而另一方面，之后出现的 XML Schema 及 RELAX NG 则可以对类型进行表述。对于 Web API 来说，这是很实用的。

在封闭环境，或试验性质的服务中，也可以省略复杂的结构描述定义，而使用形式良好的 XML 文档。不过，在公开的 Web API 中，通常都会使用合法的 XML 文档。

最近已经很少用 DTD 来书写结构描述了，一般使用的是 XML Schema 或 RELAX NG。话虽如此，如果存在多种新型的结构描述定义方式，将会导致出现多种类型的标准特殊的 XML。因此，人们开始为避免出现大量特殊的 XML 兼容标准而努力。其中，对于 Web API 来说，则出现了 Atom 这一逐渐成为 XML 默认标准的标准。

19.2.4 Atom

Atom 的最初目的是用于替代 RSS。RSS 是一种消息来源（Feed）格式的事实标准，可以用于分发 Web 站点的更新信息。RSS 最初是一种独立标准，而 Atom 则是基于 IETF 的标准。

对于消息来源而言，Atom 并不一定能取代 RSS 的地位，不过对 Web API 来说，Atom 则具有重要的作用。下面的说法或许有些混乱，Atom 一方面是一种基于（元格式）XML 的标准，而另一方面，它本身也是一种可以自扩展的元标准。

Atom 可以分为 Atom Syndication Format（rfc4287）与 Atom Publishing Protocol（rfc5023）这两种标准。

前一种标准的主要用于消息来源。后一种标准主要用于 Web 文档的更新。前者所规定的是数据格式，后者则会将符合该格式的 Atom 数据载入 HTTP，将更新数据作为一种更新指令来使用。通过 HTTP 来传递 XML 形式的指令，与接下来将会介绍的 SOAP 的功能很相似。而在 REST 的相关内容中，本章将会对此做进一步说明。

19.2.5 JSON

在 Web API 的数据格式中，与 Atom 同等重要的一种格式是 JSON。JSON 与本书的主题 JavaScript 也有密切的关联，它现在与 XML 并称为 Web API 数据格式的两大巨头。事实上，称 Web API 的数据格式将被整合至 XML 与 JSON 两者也不为过。

JSON 格式的详细信息请参见第 7 章。

19.2.6　SOAP

在此，从 Web API 通信协议的角度来回顾一下历史。可以把 HTTP 理解为 RPC（远程过程调用）。简单来讲，PRC 是跨越网络的函数（子程序）调用。Web 文档最初是作为数据来使用的，类似地，HTTP 最初是作为 RPC 使用的。

在使用 Web 时，常常会在浏览器的表单中输入值，并将其发送。在画面上可以看到的是，表单输入及按钮点击的操作，而在内部进行的则是通过 HTTP 发送该值的操作。将该值发送并等待响应的操作，可以被理解为通过 HTTP 这一通信协议，调用了远程的子程序。从 RPC 的角度来看，所发送的值就相当于传递给函数的参数。表单所支持的输入值取决于表单设计，为了实现更为广义的 RPC，必须对参数进行形式上的定义。其中具有代表性的格式是 SOAP。

■ SOAP 与 RPC

SOAP 是基于其前身，XML-RPC 这一自定义标准所定义的一种标准标准[①]。SOAP 通过 XML 来表述参数。用一种可能会产生歧义的方式来说的话，SOAP 是一种通过 XML 来表述命令的 RPC。SOAP 是独立传输的，可以对 XML 所表述的命令在网络中的传输方式进行自由设定。不过在实际中，一般都默认使用 HTTP 传输，因此本书将以此为前提。也就是说，SOAP 通过 HTTP 来传输 XML 所表述的指令，并以此实现 RPC。

2000 年前后，SOAP 与 Web 服务这一术语一起被广为宣传。由于宣传力度很大，SOAP 几乎被等同于了 Web 服务。由于这一影响至今仍然存在，因此一些没有使用 SOAP 的 Web API，会避免使用 Web 服务这一术语。

存在大量与 SOAP 与 Web 服务相关的标准。这里我们仅介绍其中最为重要的两种标准——WSDL 与 UDDI。WSDL 是对接口进行定义的描述语言。如果说 SOAP 是用于 RPC 调用的标准，WSDL 的作用则是对 RPC 的格式（参数及返回值）进行描述。UDDI 是用于 WSDL 检索目录的描述语言。

19.2.7　REST

仅从形式上来看，Atom Publishing Protocol 与 SOAP 是相似的，它们都会通过 HTTP 传递 XML 形式的指令。不过，这样的想法是不正确的。Atom 基于的是 REST 的理念，而 REST 可以说和 RPC 是相悖的。这里先将 REST 与 SOAP 进行比较以说明其不同，之后还将再次说明一下 REST。

■ 与 SOAP 的比较

SOAP 基于的是 RPC 的理念，以调用远程子程序[②]。因此，HTTP 上所传输的 XML 被认为是指令。

而另一方面，根据 REST 的理念，在 HTTP 上传输的数据。也可以将这里的数据替换为文档或资源。对于 REST 来说，服务器上的资源或用于更新文档的更新信息，都是通过 HTTP 传递的。而 HTTP 方法（GET 及 POST）则相当于指令。也就是说，在 Atom 中所写的信息并不是指令，而是用于更新的数据。19.2.4 节使用了 XML 形式的指令这一说法，严格来讲，这是不正确的。

19.2.8　简单总结

让我们简单回顾一下 Web API 的历史。Web API 是 Web 服务的调用规则。目前 Web API 的发展趋势

① 不过，由于 SOAP 标准过于复杂，也有一些 Web API 仍然在使用 XML-RPC。此外还有一种使用 JSON 执行功能的 JSON-RPC，它同时继承了 XML-RPC 的语义。

② 以面向对象的方式来解释远程子程序的话，他们使用的是远程对象。SOAP 所调用的是远程对象的方法。广义上来说，远程对象也是一种 RPC，所以本书将以 RPC 的方式来说明 SOAP。

是，XML 与 JSON 逐渐成为了标准的数据格式。此外，其通信协议主要分为基于 SOAP 与基于 REST 的两大类型。基于 SOAP 的协议与 XML 关系密切，而基于 REST 的协议则与 XML 中的 Atom 或 JSON 相关。

19.3 Web API 的组成

19.3.1 Web API 的形式

表 19.1 总结了 Web API 的形式的演进，下一章的实例也会再次提及这一内容。不过，HTTP API 与语言 API 是本书自定义的术语，对此请加以注意[①]。

表 19.1 Web API 的演进

名称	说明
Web 抓取	非官方手段
HTTP API	定义了请求 URL 及响应数据的形式
语言 API	定义了函数及类库（JavaScript API、JavaScript 库）
微件 API	HTML 的代码片段

■ HTTP API 与语言 API

直至 HTTP 级别的标准制定之后，Web API 的历史才真正开始。从开发者的角度来看，使用 HTTP API 即是一种网络程序设计。HTTP 的查询参数需要自己构建，响应也需要自己来分析。

在能够对 HTTP 的详细信息进行函数调用，且出现了能够隐藏类的库之后，Web API 发展至了第二阶段。在这类语言 API 中，有一些是 Web API 的提供者（服务提供者）自己提供的，也有一些是无关的开发者自己制作的第三方 API。即使 Web API 的提供者只提供了 HTTP API，如果该 Web API 得到了广泛使用，则通常会有第三方为其制作语言 API。第三方制作的 API 虽然是非官方的，但如果成为了主流，也就成为了事实上的标准库。此外，就算存在官方的语言 API，提供者也无法为世上所有的程序设计语言都提供 API。因此，非主流语言的语言 API 还是需要借助第三方的非官方 API。

即使存在官方的语言 API，其 HTTP 级别的 API 也可能没有公开。这里的没有公开，并不是说它们是一种机密。毕竟 HTTP 通信本身就没有什么机密内容。这里指的是，这些 API 没有将其 HTTP 级别的格式作为一种标准标准公布于众。

不公开 HTTP 级别的 API 的好处在于，服务提供者的自由度将会提高。一旦公开了 HTTP API，就难以再更改。而如果不公开，即使更改了 HTTP 级别的通信，也能通过语言 API 来处理这一更改。

■ JavaScript API 与微件 API

在语言 API 所支持的语言中，JavaScript 并不是主流。当前，在使用 Web API 时，从服务器端调用是主流做法，之后一节也会对此进行说明。不过渐渐地，在客户端，也就是浏览器中调用 Web API 的情况也越来越多了。客户端 JavaScript 所使用的语言 API 或微件形式的 API 将会逐渐成为发展方向。在有些地区，微件也被称为博客插件（blog part）[②]。也有人将其称为小工具或插件[③]。

只要将特定功能的代码片段复制粘贴至 HTML 内，即可使用微件。在微件中隐藏有语言 API，其中进一步隐藏了 HTTP API。用户不需要知道其详细的内部实现就能够使用。从这一点上来说，微件是最高层次的 API。

① 正如之前小节所讲，Web 抓取是一种可以以任意方式对公开的 HTML 及 HTTP 通信进行分析的方式。如果目标 HTML 的结构发生了改变，则必须重新书写。它虽然无法被称为是一种 Web API，但仍然作为 Web API 的早期历史而被记录在了表内。

② 在很多博客模板中，都提供了类似于微件的功能，故而得名。——译者注

③ 微件一词也有 gadget、plugin 等名称。这里统一使用微件这一译法。——译者注

Google 或 Amazon 的搜索框这类的微件类型，很早以前就存在了。最近还出现了不使用 iframe 的形式，以及可以调用用于用户验证的 API 的微件 API。目前流行的 Web API 多以这种方式发展。下面是一些具体的实际例子，在之后的章节中还将介绍。

- Facebook Social Plugin
- Twitter @anywhere
- Google Friend Connect（GFC）

19.3.2　Web API 的使用

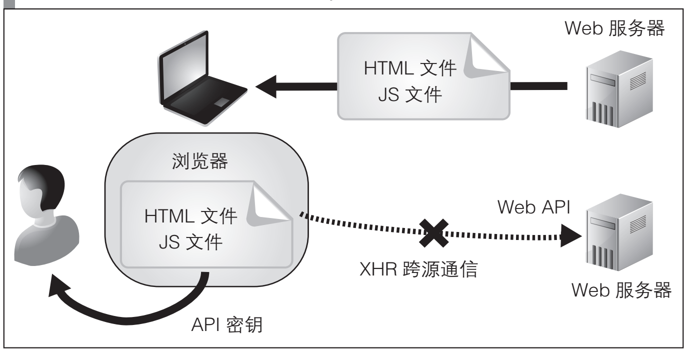
根据 Web API 的调用位置的不同，可以按照以下方式分类：

- 在服务器（Web 程序）中调用
- 在浏览器中调用
- 在原生应用程序中调用（桌面程序或智能手机应用）

当前的主流做法是在服务器中调用 Web API。而这一部分的主题，客户端 JavaScript，并不是主流的 Web API 调用方式。其理由之一是，在通过客户端 JavaScript 调用 Web API 时，存在以下问题（图 19.1）。

- 跨源限制
- 无法对 Web API 的调用加密（API 密钥等）

图 19.1　在通过客户端 JavaScript 调用 Web API 时存在的问题



关于跨源限制的问题，第 3 部分已经进行了说明。本节将再次总结一下该问题的主要的解决方法。详细的内容请参见第 3 部分，以及下一章中的实际例子。

- 通过服务器中转（代理）
- JSONP
- XMLHttpRequest2（CORS：Cross-Origin Resource Sharing）
- 框架（iframe）间的协作
- PostMessage
- 跨源接收者

19.3.3　RESTful API

非 SOAP 类型的 Web API 统称为 REST API 或 RESTful API（REST 式的 API）。既非 SOAP 又非 RESTful 的 Web API 定义方式也有很多，因此，这种称法从理论上来说是不正确的。不过这种称法已经被广为使用，所以本书也将使用这一方式。

■ REST 的定义

REST 这一术语的出现，与 Web API 并没有什么关系。REST 是 Representational State Transfer 的缩写，是由 HTTP 标准的其中一位制定者罗伊 · 菲尔丁（Roy Fielding）在其论文中使用的自造词。REST 这一术语并不指代某一特定的技术，该术语的出现，是为了在分析 Web 架构时，指代 Web 所具有的分布式系统模式这一特征。也就是说，首先是有了 Web 这一实际的系统，之后才通过论文对如何扩放 Web 进行分析的。之后，这一模式被命名为 REST，而 Web 是一种遵循了 REST 模式的分布式系统，从而得到了能够对其进行扩放的结论。

在 REST 中，分布式系统中的对象被称为资源。如果通过 URI 这一命名空间来访问资源，则能使其以某种特定的形式实例化。试考虑实际的 Web 的情况，我们可以理解为，浏览器访问了服务器之后，服务器端所管理的数据被以指定的格式（XML 或 JSON 等）返回至浏览器。可以通过 HTTP 的 GET、POST、PUT、DELETE 方法来对资源进行获取、创建、更新及删除操作。

REST 的特征是只能够执行资源的更新这一种远程操作。在 REST 之前，分布式系统的主流选择是 RPC。SOAP 或 CORBA 等分布式对象也属于 RPC 的分支。这些 RPC 的核心是，对分布式系统进行子程序调用，或者对远程对象进行方法调用。粒度较大的面向服务架构也属于这一类型。对于这一类型，关键的问题在于执行的是何种操作。

而另一方面，REST 的核心是资源。有时 REST 也被称为面向资源或面向文档的方式。它所允许的操作就只有对资源（文档）的更新（包含了创建、删除在内的，广义的更新）。这是一种很强的限制条件。不过，正是有了限制条件，才能决定一种架构。提出了 REST 的这篇论文的关键结论是，REST 这一限制条件，是 Web 具有可扩放性的根基。

在正确理解了 REST 之后，我们就能发现将非 SOAP 的类型称为 REST 是错误的。如果非要将分布式系统分为两种类型，那么 RPC 系和 REST 系这样的术语才是最合理且恰当的。前者侧重于操作，而后者则侧重于资源的更新。对于通过非 SOAP 来描述 REST 的情况，可以将其理解为，选择了 SOAP 这一 RPC 系中的代表类型来说明。

■ SOAP 与 REST

从实际使用方式上对 SOAP 与 REST 做比较的话，会发现两者最大的区别在于 URL 的命名风格。SOAP 的 URL 命名是与操作相对应的。因此，URL 的命名是动词风格的。以面向对象的方式将对象名与操作名相结合之后，得到的 URL 是名词与动词的组合。

另外，REST 的 URL 命名是与资源相对应的。因此，URL 的命名是名词风格的。在 URL 中没有动词，因为动词已经通过 HTTP 的 GET 或 POST 进行指定。正因如此，也有人将 RESTful API 理解为一种 URL 的命名风格。这种观点有些狭隘，不过从实际使用的角度来看，也是一种对 RESTful 的妥当的定义。

有很多 Web API 既没有遵循 SOAP 风格，也没有遵循 REST 风格。不过在实际中，一种 Web API 即使没有遵循 REST 风格，只要它不是 SOAP，也会被命名为 RESTful API。SOAP 这一标准标准庞大且复杂，非常规整，而与之相对的 RESTful 则被认为是其对立面。这是上述情况产生的深层原因之一。因此，对于很多 Web API 来说，只要它不是 SOAP 风格，那么无论有没有遵循 REST 的 URL 命名风格，都会自称为 RESTful API。

19.3.4 API 密钥

一些 Web 服务会发行一种被称为 API 密钥的密钥，以使用其 Web API。在使用 Web API 时，需要提交 API 密钥。从 HTTP 级别来看，就是在发送请求时通过查询参数来传递 API 密钥，服务器断将检查其值。API 密钥的主要作用如下所示。

- 使用限制（API 的调用次数等）
- （将来可能的）收费

如今很多的 Web API 都有使用限制。不过，据笔者所知，目前还没有收费的 Web API。因此，事实上 API 密钥是免费的。虽然也有些服务通过与邮件地址相关联来制约 API 密钥无限制的发行，但也不是什么严格的限制。

由于这样的背景，如果将现在的 API 密钥理解为机密信息，也不是很合适。要是窃取了 API 密钥并随意使用的话，就会引发使用限制，于是自己的程序可能会无法再调用 Web API。话虽如此，这样的情况也没有什么经济损失，最多是让人恼火而已。

API 密钥是否需要加密，对于客户端 JavaScript 来说具有重要意义。这是因为，理论上，在客户端 JavaScript 中使用 Web API 时，无法对 API 密钥加密。

因为通常 API 密钥都会被写在 JavaScript 代码中，所以用户很容易就可以通过浏览器对其进行读取。即使花功夫对 JavaScript 代码进行了混淆处理，只要用户能通过浏览器调用 Web API，就能够轻松地获知 HTTP 通信的情况。因此，如果有必要对 API 密钥加密，则不能以客户端 JavaScript 来调用 Web API。

19.4 用户验证与授权

19.4.1 Web 应用程序的会话管理

如果 Web API 获取的是任何人都可以阅读的文档，则不必进行用户验证。不过，如果 Web API 会对文档进行更新，或者获取的是具有访问限制的文档的话，就有必要进行用户验证。

■ 用户验证机制

为了理解 Web API 的用户验证，我们首先要说明依稀通常的 Web 应用中的用户验证机制。归根结底，Web 应用的用户验证机制是一种会话管理。如果直截了当地对 Web 应用会话管理的定义进行说明，可以说这是一种用于判断 HTTP 请求是由谁发送的机制。对于 HTTP 这一通信协议来说，1 次请求与 1 次相应组成了一个单位。即使是同一个用户通过同一个浏览器向同一个服务器发送了请求，这一请求（在原则上）与其前一个请求之间也不会有相关性。因此，为了识别出由同一个用户发出的请求，需要对会话进行管理。

对于每一个独立的 HTTP 请求而言，需要对发送者相同的请求添加标记。基于这些标记，服务器端就能判断请求的发送者。可以将 Cookie 或特殊的查询参数用作区分请求的标记。

在 Web 应用的服务器端以用户为单位所保存的状态被称为会话。通过在会话中保存用户的信息，就能够根据相应的用户来返回响应。

■ Cookie 与会话管理

这里略去对使用查询参数的方式的说明，仅对 Cookie 进行说明。Cookie 实际上是名为 Cookie 的 HTTP 请求头部信息。Cookie 头部信息与其他的请求头部信息的不同之处在于，（支持 Cookie）的浏览器将会把 Cookie 头部信息的值保存在本地。

浏览器将会记忆接收自服务器的 Cookie 值，如果请求是发送给同一个服务器的，所保存的 Cookie 值就会被添加至请求中[①]。Web 应用将会引用所收到的 Cookie 头部信息，以判断该请求是来自于哪一个浏览器的。

Cookie 只能区分是否是同一个浏览器，对于不同的用户使用了同一个浏览器的情况是无法分辨的。在企业或学校中，这已经是相当危险了，如果是网吧这样的同一台电脑会有大量不固定的用户的情况，则会有致命的安全风险。而且，如果 Cookie 头部信息的值成为了用于个人识别的重要依据，Cookie 值将会是

① 服务器会通过发送至客户端的 Set-Cookie 响应头部信息来传送 Cookie 值。虽然也可以通过 JavaScript 来生成 Cookie 值，不过通常在会话管理中不会这么做。

一种比密码更为机密的信息。然而在普通的浏览器中，保存于本地的 Cookie 值并没有严格的保护功能。

为了防范这一风险，在 Cookie 值中加入了 Web 应用所发布的会话 ID。会话 ID 起到的是获取 Web 应用的会话信息的密钥的作用。会话 ID 通常是基于乱数生成的值，本身并没有什么意义。在用户登出 Web 应用时，或者用户在一定时间内没有发出请求时，Web 应用就会清除会话 ID。通过限定 Web 应用中会话 ID 的有效时间，就能够减少 Cookie 值，即会话 ID 被盗时的风险。通常，这一有效期限被称为"会话超时时间"。

总结来说，本身不具有含义的会话 ID 被作 Cookie，在浏览器与 Web 应用之间传递。浏览器将对向哪一个服务器传递哪一个 Cookie 值进行管理，Web 应用会对所接收到的会话 ID 所能引用的会话信息进行管理。两者通过这样的状态管理，实现了对 Web 应用的用户登录状态进行管理。

■ Cookie 的有效期限

同时，还存在会话 Cookie 这一容易引起混淆的术语。这一术语和 Web 应用中的会话并没有关系，它表示的是浏览器的启动状态。没有明确指定有效期限的 Cookie 的有效期仅限于浏览器进程存在的范围内。也就是说，一旦浏览器关闭，Cookie 就无效了。

而在指定了 Cookie 的有效期限后，它就会被保存在浏览器所在的本地磁盘中，在浏览器重启后 Cookie 也有效。前面那样的随着浏览器进程的结束而失效的 Cookie 俗称为会话 Cookie。这很容易引起混淆，还请加以注意。

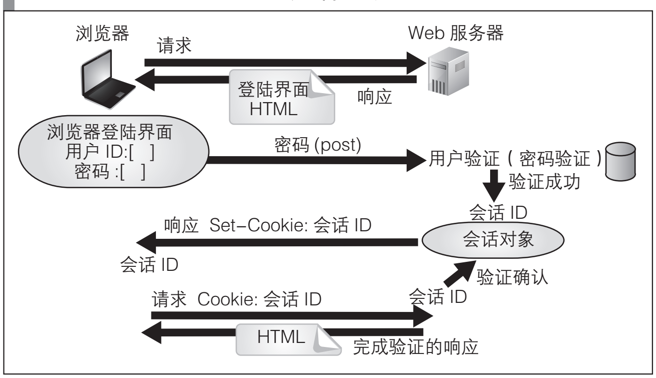
19.4.2 会话管理与用户验证

下面通过图片来说明，通过 Cookie 和会话管理进行 Web 应用的用户验证的一般流程（图 19.2）。其会话未被管理的用户在访问需要登录的 Web 应用时，将会出现登陆画面。用户需要在登陆画面输入用户 ID 与密码。

Web 应用在接收了用户 ID 与密码之后，会通过内部的数据库或目录验证密码。如果验证成功，就会创建用于管理登录状态的会话，并将会话 ID 作为 Cookie 值返回。

之后，在 Cookie 的有效期限及会话的有效期限内，用户就将处于 Web 应用的登录状态。

图 19.2 通过 Cookie 进行会话管理的概念图



除了下一节中将要考虑的 Web API 与 Cookie 的使用规则之外，还需要记住一些重要的规则。即 Cookie 仅能够被发送至其发布者的 Web 应用。

浏览器会将 Cookie 发送至 Cookie 发布者的 Web 服务器中，对于发送至其他 Web 服务器的请求，则不会发送该 Cookie[①]。这一限制与跨源限制相类似，不过，跨源限制尚且有些回避方法，而 Cookie 值则完

① 严格来说，还是可以对一些细节进行控制的，不过从整体上来看，只要这样理解就没有问题了。

全不可能被发送至其他的服务器。

19.4.3 Web API 与权限

接着上一节的内容，我们再来考虑一下 Web API 与用户验证的问题。在使用 Web API 时，存在 Web API 的服务器、使用了 Web API 的应用，以及在浏览器中访问该应用的用户这三者。对于这些术语的说明还很混乱，目前还不存在完全统一的命名。在本书中，将提供 Web API 的服务器称为服务提供者（Service Provider，SP），将使用了 Web API 的应用称为第三方应用，将对浏览器进行操作的使用者称为用户。

基本上，可以将服务提供者认为是 Google、Facebook 或 Twitter 等保存有大量用户账户的服务。第三方应用则是使用了由这些服务提供者所提供的 Web API 的 Web 应用。用户持有服务提供者的账户，并通过浏览器访问并使用第三方应用。

第三方应用在使用服务提供者的 Web API 时，可以分为通过服务器端调用及通过客户端调用这两种形式。其流程分别如下图所示（图 19.3 与图 19.4）。

■ 服务器端的 Web API 调用及其权限

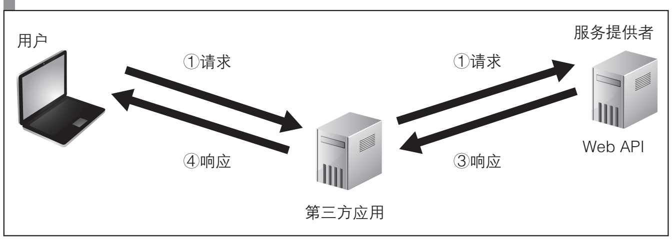
在通过服务器端调用 Web API 时，由于没有跨源限制等麻烦的问题，因此实现起来较为容易。与从客户端调用 Web API 相比，这时，服务器充当了用户调用 Web API 时的中转站。由于是中转处理，因此可能会有丢失响应的风险。同时，由于 Web API 的调用处理被集中于服务器，因此还可能会出现性能瓶颈。不过，这种方式也有一些优点，比如可以缓存相同 Web API 的调用结果，还可以整合调用过程。综合考虑所有这些情况，通过服务器调用或许是一种更有效率的方式。

在通过服务器端调用 Web API 时，需要考虑 Web API 调用者的权限。理想的情况是，用户通过服务提供者提供的账户，以该账户所具有的权限调用 Web API。不过，正如前一节中所讲，Cookie 只能被发送至其发布者的服务器。因此，与服务提供者的登录状态相关的 Cookie 无法被发送至第三方应用。

在 OAuth 或类似的协议出现之前，用户需要将其在服务提供者上的账户（用户 ID 及密码）告诉第三方应用，由第三方应用以模拟的形式登录服务提供者。这虽然也是一种解决方案，却不够安全。

为了解决这一问题，出现了 OAuth 这一授权协议，本节接下来将对其进行说明。从用户处获得授权的第三方应用，将会通过用户的权限对服务提供者的 Web API 进行调用。

图 19.3 通过服务器端调用 Web API

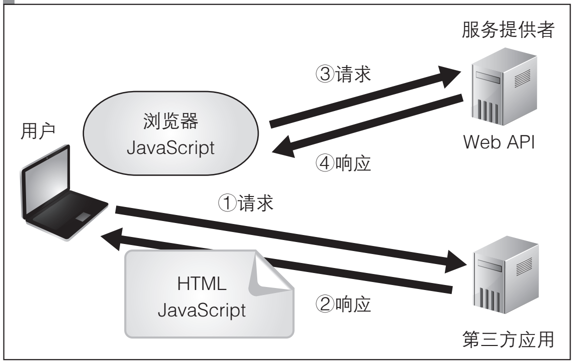


■ 客户端的 Web API 调用及其权限

在通过客户端 JavaScript 调用 Web API 时，同样有一些与权限相关的问题，不过这时的情况稍有不同。

如图 19.4 所示，在调用 Web API 时，浏览器将会发送 HTTP 请求给服务提供者。与通过服务器端调用时的情况不同，该请求中包含了与用户在服务提供者上的登录信息相关的 Cookie。乍一看，这样就避免了权限问题，但可惜的是，这时仍然存在着问题。这一问题我将在之后的 CSRF 中进行说明，而该问题的解决方法将在 OAuth 中进行说明。

图 19.4 通过客户端调用 Web API



■ CSRF

在用户完成了服务提供者的登录之后，浏览器向服务提供者发送的 HTTP 请求就会变成已登录的状态。乍一看，如图 19.4，通过客户端调用 Web API 的方式，解决了用户验证的问题。但事实上，这其实是一个安全漏洞。

这是因为，举个极端的例子来说，如果 Web API 提供了文档创建或文档删除的功能，只要打开 Web 页面，就能在不知不觉中随意执行文档删除操作。这是一种被称为 CSRD（Cross-Site Request Forgeries）的攻击方式。如果存在这样的 Web API，将会是一种致命的安全隐患。对用户调用 Web API 的请求进行许可，才是授权机制应有的功能。为了实现这一目的，我们可以采用之后将会介绍的 OAuth 2.0 这一用户代理流程。

19.4.4 验证与授权

在说明 OAuth 的授权机制之前，我们首先整理一下验证（authentication）与授权（authorization）相关的术语。这些术语比较容易混淆，因此，表 19.2 对其进行了总结。

表 19.2 验证与授权

名称	说明
验证	为了对身份进行判断，而对个人或进程所提供的资格信息进行验证
授权	将可以执行某些操作，或者可以使用某些位置的权限授予给个人

Web 应用中的验证实际上指的是登录处理。在输入了用户 ID 及密码之后，就能够登录系统。密码是只有本人才知道的机密信息。也就是说，这里假定只有本人知道这一机密信息。

除了密码，还有一些其他的只有本人才知道（或拥有）的机密信息，例如生物验证或密钥验证公钥验证（PKI 验证）等。不过对于 Web 应用来说，密码验证是事实上的标准。如前节所述，在登录过程中，是通过以 Cookie 发送会话 ID 的方式来进行用户验证的。即会话 ID 起到了临时机密信息的作用。

授权则决定了用户在登录之后可以进行哪些操作。例如，改写自己所创建的文档的权限、读取他人所创建的文档的权限，以及向文档中添加注释的权限等。

以上这些对验证与授权的说明，只有在某个 Web 应用（服务）与用户之间，是一对一处理的前提下才有意义。前一节一开始也提到过，在 Web API 的使用过程中，含有服务提供者、第三方应用与用户三者。对于这三者之间的验证与授权，需要有另一种类似于权限委托的机制。这一机制被称为分布式验证系统 / 协议或分布式授权系统 / 协议。

分布式验证的例子有 OpenID，分布式授权的例子有 OAuth。在 Web API 领域，分布式授权系统正在发挥越来越重要的作用，本章接下来将对 OAuth 进行说明。

19.4.5　OAuth

关于 OAuth 的详细信息，请参见下面的 URL。IETF 正在制订 OAuth 2.0 的标准，现在已经有一部分 Web API 对 OAuth 2.0 提供了支持。

http://oauth.net

OAuth 是一种授权传递协议。也就是说，这一协议标准的作用是将在服务 A 中授予的权限传递给服务 B。有时 OAuth 也被称为验证协议或授权协议，不过这种称法可能会引起混淆。如果将其称为验证协议，则可能让人误以为它是一种类似于 BASIC 验证或表单验证等的协议，需要通过发送用户 ID 及密码来进行。OAuth 是对已授权的权限的转让。因此，称为权限转让协议也没有问题。

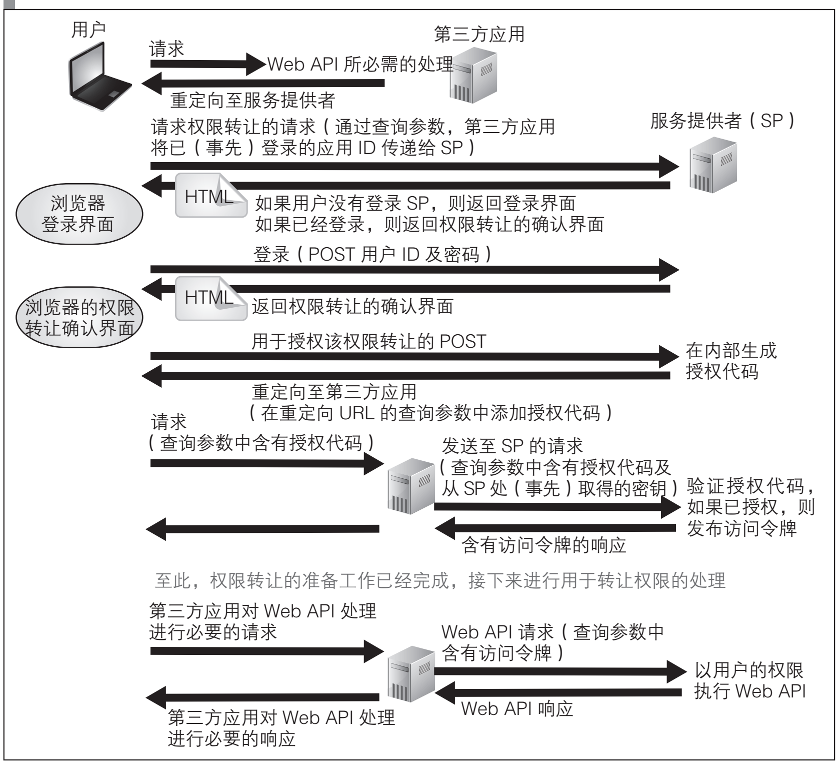
用 Web API 领域的术语来说的话，OAuth 的作用是将服务提供者中的用户权限（创建文档或添加注释等）转让给第三方应用。这时，不必将用户与服务提供者之间的机密信息（密码及会话 ID 的 Cookie 值）传递给第三方应用，就能实现权限的转让。除了 OAuth，也有其他一些类似的自定义协议，用于转让权限。不过 OAuth 2.0 正渐渐成为 Web API 领域的主流。

通常来说，OAuth 被用于图 19.3 这样的，从服务器端调用 Web API 的情况。不过在 OAuth 2.0 中增加了新的用户代理流程，在图 19.4 这样的客户端 JavaScript 中也能使用。

■ OAuth 2.0 的服务器端流程

首先对通常的 OAuth 流程进行说明。图 19.3 中的基本场景为，第三方应用通过用户的权限来调用服务提供者的 Web API。这时，其内部的通信协议概况如图 19.5 所示。

图 19.5　OAuth 2.0 的流程



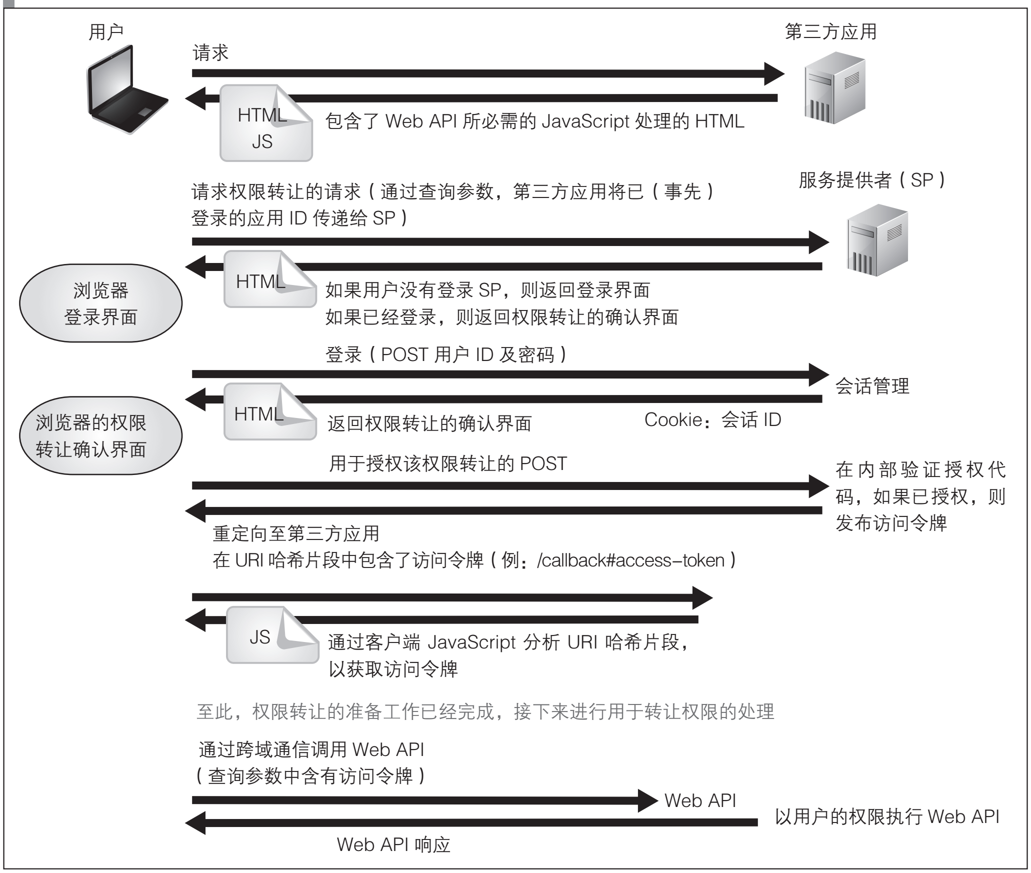
OAuth 的详细信息可以在其标准书或相关的专业书籍中找到。图 19.5 中的流程的目的很明确，就是为了获取访问令牌。所谓访问令牌，是一种具有时效限制的机密信息的替代品。在调用具有访问令牌的 Web API 时（通常是在调用 Web API 时，通过 HTTP 查询参数来传递令牌的），将会以服务提供者中的用户账户所具有的权限来执行操作。

■ OAuth 2.0 的用户代理流程

本章的主题是客户端 JavaScipt，如果要通过客户端 JavaScript 来调用 Web API（图 19.4），则可以使用 OAuth 2.0 的用户代理流程（在现在的 OAuth 2.0 标准中，它被称为隐式授权）。此时，其内部通信协议的概况如图 19.6 所示。

与图 19.5 不同的是，这种情况下第三方应用（服务器端）没有与服务提供者进行通信。如果用户与服务提供者之间的 HTTP 通信处于登录状态，就会通过 Cookie 进行会话管理，对此请加以注意。图 19.6 中的通信的目的也是为了获取访问令牌。该访问令牌的作用与图 19.5 中的相同。

图 19.6 OAuth 2.0 的用户代理流程



第 20 章 Web API 的实例

本章将介绍 Web API 的实例。最近，主流的 Web 应用都提供了 Web API，构建第三方应用及其软件生态环境也成为了一种常见做法。之后我们将通过一些实例，来感受一下 Web API 的共性并思考其未来可能的发展方向。

20.1 Web API 的分类

下面是一张具有代表性的 Web API 的分类，引自 ProgrammableWeb[①]（表 20.1）。通过它，我们可以对 Web API 的实际情况有一个把握。这份分类能反映的不过是当前的 Web API 的情况[②]。在 Web 中充满了可能，或许在今后还会出现现在所没有的新的类型。因此，不要认为表 20.1 列出的就是 Web API 所有的应用实例了[③]。

表 20.1　ProgrammableWeb 的分类摘要

分类	说明
Advertising	广告。Google 最强
Answers	社会化问答。Stack Overflow 与 Quora 等正在追赶 Yahoo Answers
Blog Search	博客搜索。在 Technorati 之后不再那么热门
Blogging	博客。由于被 Twitter 压制，而不再那么热门
Bookmarks	社会化书签。在 del.icio.us 之后不再那么热门
Calendar	日历。Google Calendar 最强
Chat	聊天。由于被 Twitter、Facebook 压制，而不再那么热门
Database	数据库。可以将 NoSQL 这类 RDB 的 CURD 操作转化为 REST 操作等
Dictionary	字典
Email	Web 邮件
Enterprise	salesforce.com 最强
Events	区域事件分享。Eventful 与 Upcoming.org 为最强的两家
Feeds	消息来源
File Sharing	在线文件共享
Financial	股票信息、外汇汇率等
Food	餐厅搜索等
Games	游戏。在 SecondLife 之后不再那么热门
Internet	其实这一分类指的是其他类型。诸如 Amazon EC2 等
Job Search	求职搜索
Mapping	地图。Google Maps 最强
Media Management	BBC 的归档功能为其中一强

① http://www.programmableweb.com/

② 这份分类是基于美国的互联网现状总结出来的，和国内的情况可能略有差异。——译者注

③ 表 20.1 的说明包含了执笔本书时笔者自己的一些感想。

（续）

分类	说明
Messaging	消息收发。诸如 411Sync 等
Music	Last.fm 最强
News	诸如 Digg 及 Reddit 等
Office	诸如 Google Docs、SlideShare、Zoho 等
Payment	PayPal 为其中一强
Photos	照片分享。Flickr 最强，TwitPic、Smugmug、Instagram 紧随其后
Project Management	项目管理。诸如 Basecamp 等
Real Estate	地产搜索服务。诸如 Zillow 等
Recommendations	推荐搜索服务。诸如 Yelp 等
Reference	诸如 GeoNames 及 Wikipedia 等
Search	搜索。Google、Yahoo! 与 Bing 为最强的三家
Shipping	FedEx 等的配送服务
Shopping	在线购物。Amazon 与 eBay 是最大的两强，而新兴的 Groupon 等团购服务正在与其激烈竞争
Social	诸如 Twitter、Facebook、Foursquare、LinkedIn、MySpace 等
Storage	在线存储。Amazon S3 是最领先的，此外还有 Dropbox 等新兴服务
Telephony	互联网电话。Twilio、Skype 等
Tools	Google App Engine 等
Utility	Google Translate、Evernote 等
Video	YouTube 最强
Weather	天气预报

20.2 Google Translate API

Google Translate API 是一种翻译 Web API，在通过该 API 发送文本之后，将会返回翻译后的文本。由于它的功能很简单，因此能够更好的分析其 Web API 的本质。不过可惜的是，Google Translate API 服务今后将会被终止[①]。

虽然 Google Translate API 将会终止提供服务，不过可以将其作为基础，对其他的 Web API 进行介绍与说明。与 Google Translate API 自身的用法相比，更应关注其作为 Web API 的使用方式。

本节之后将基于 Google Translate API v2 进行说明。大家可以通过下面的地址获取 API 的参考文档。

http://code.google.com/intl/en/apis/language/translate/overview.html

除了 Google Translate API，还有另一个名为 Google Translate 的 Web 应用。可以通过下面的 URL 访问。

http://translate.google.com

由于 Google Translate API 所提供的功能和其 Web 应用的功能基本相同，因此也可通过前一章介绍的 Web 抓取方式，对 HTTP 通信及 HTML 进行解析，以在程序中对其进行访问。

理论上也能够开发一个程序，在发送文本后获取翻译结果，不过，这种方式的效率不会很高，而且有可能在 HTML 的结构发生改变时无法正常运行。其实所实现的功能相同，使用 Web API 也有其优势。在使用 Web API 时不需要随 Web 应用的标准调整而更改[②]。

① http://googlecode.blogspot.com/2011/05/spring-cleaning-for-some-of-our-apis.html

② 不过颇具讽刺意味的是，如果 API 的提供被终止，则仍会受到影响。当然，如果 Web 应用被关闭，也会产生这一问题，因此这也算是一种无法避免的风险了。

20.2.1　准备

如果要使用 Google Translate API，必须拥有 Google 账户。在获取了 Google 账户之后，需要登录 Google 的站点。账户是可以免费获取的。访问以下页面，就能够进入 Google 所提供的 Web API 的管理界面。

http://code.google.com/apis/console/

在管理界面中启用 Google Translate API 服务之后，就能够获取其 API 密钥。API 密钥的作用是限制每天调用 Web API 的次数。Google 账户是免费的，几乎可以获取无限多个账户，所以这也算不上严格的限制。

20.2.2　执行方式的概要

Google Translate API 的基本执行方式为，向下面的 URL 发送 HTTP 请求，并接收 JSON 形式的响应。

```
http://www.googleapis.com/language/translate/v2? 查询参数
```

URL 路径是固定的，请求中可以改变的只有查询参数部分。与函数调用相比，可以认为 URL 路径就相当于函数名，查询参数则是参数，而响应则相当于函数的返回值。对于只听说过 Web API 这个词而没有实际见过 Web API 的人来说，当发现它其实如此简单时或许会有些失望吧。最基本的 Web API 的形式确实就是如此。

在 Web API 的标准中规定了可以使用的查询参数名。只要掌握了表 20.2 中所列的查询参数，就能够对其进行最为基本的使用了。更详细的内容请参见其他参考文档。

表 20.2　URL 的查询参数

参数名	是否必须	说明
q	O	指定需要翻译的字符串
key	O	指定 API 密钥
source	X	设定所要翻译的原字符串的语言（如果没有指定，则会自动判断）
target	O	指定希望翻译成哪种语言
callback	X	指定 JSONP 所需的回调函数（之后详述）

在通过程序调用之前，请先直接通过 HTTP 方式对功能进行确认。可以通过 curl 指令来确认其执行情况。curl 是可以在命令行中使用的 HTTP 客户端工具。它会将 HTTP 请求发送至参数所指定的请求 URL，并以标准方式输出 HTTP 响应。

先不必在意请求 URL 的路径、参数及 API 密钥等详细信息，只要关注所返回的响应的格式即可（图 20.1）。

图 20.1　确认 Google Translate API 的功能

```
$ export APIKEY= 所获取的 API 密钥

# 参数 key、target 及 q 是必需的（参数 source 可以被省略）
$ curl "http://www.googleapis.com/language/translate/v2?key=${APIKEY}&q=I%20found
%20a%20cat&target=zh-CN&source=en"
{
    "data": {
        "translations": [
            {
                "translatedText": " 我发现了一个猫。"[1]
            }
```

① 读者可能会觉得这句话不太通顺，不过，Google Translate API 的翻译结果确实如此。——译者注

```
            ]
        }
    }
```

图 20.1 中的响应是 JSON 格式的（参见第 2 部分 7.2 节）。对于服务器端的情况，可以在代码中对 HTTP 通信及 JSON 进行分析，以使用 Google Translate API。不过对于客户端 JavaScript 的情况，则会产生跨源限制，无法直接使用（参见前一章）。但 Google Translate API 也对 JSONP 提供了支持，因此，能够回避跨源限制这一问题。

支持 JSONP 的 Web API 如果要接收 JSONP 格式的响应，则需要使用特定的调用方式。对于 Google Translate API 来说，需要像图 20.2 那样将 callback 参数传递至请求 URL。callback 参数的值所指定的名称将被作为函数名，并在之后返回一个 JSONP 格式的响应。

在很多支持 JSONP 的 Web API 中都使用了 callback 这一参数名称，它已经成为了一种事实上的标准。

图 20.2 在请求中使用 JSONP

```
$ export APIKEY= 所获取的 API 密钥
$ curl "http://www.googleapis.com/language/translate/v2?key=${APIKEY}&q=cat&target=
zh-CN&source=en&callback=myfunc"
myfunc({
    "data": {
        "translations": [
            {
                "translatedText": " 猫 "
            }
        ]
    }
}
);
```

20.2.3 使用了 Web API 的代码示例

代码清单 20.1 是一个在 JavaScript 中使用前节所述的 JSONP 的示例。_YOUR_APIKEY_ 部分需要替换为所获取的 API 密钥。

代码清单 20.1 是一段 HTML 文件的片段。从现在起，本章出现的 HTML 代码都将只是片段，对此请加以注意。在代码清单 20.1 中，为了更清楚的体现 JSONP 的执行方式，而有意省略了库的部分。通常，为了使 JSONP 能正常工作，需要创建 script 标签。

代码清单 20.1 使用了 JSONP 的代码示例

```
<script>
// JSONP 的回调函数
function translateText(response) {
    alert(response.data.translations[0].translatedText); // 显示翻译结果
}

function doTranslate() {
    var newScript = document.createElement('script');
    newScript.type = 'text/javascript';
    var sourceText = encodeURIComponent(document.getElementById("sourceText").value);
    var source = 'http://www.googleapis.com/language/translate/v2?key=_YOUR_APIKEY_
    &source=en&target=zh-CN&callback=translateText&q=' + sourceText;
    newScript.src = source;

    // 为了能够调用 JSONP，需要动态创建 script 标签
    document.getElementsByTagName('head')[0].appendChild(newScript);
}
</script>
```

```
<input type="text" id="sourceText" />
<div onclick="doTranslate()">翻译</div>
```

有一些 JavaScript 库可以隐藏 JSONP 的调用代码（诸如动态创建 script 标签等操作）。jQuery 就是其中之一。代码清单 20.2 是一个通过 jQuery 来使用 Google Translate API 的代码示例。

代码清单 20.2　通过 jQuery 来使用 JSONP 的代码示例

```
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.6.1/jquery.min.js"></script>
<script type="text/javascript">
function doTranslate() {
    $.ajax({
        'type':'GET',
        'url':'https://www.googleapis.com/language/translate/v2',
        'data':' {key:'_YOUR_APIKEY_', q:$('#sourceText').val(), target:'zh-CH'},
        'dataType':'jsonp',                  // 将数据类型指定为 JSONP
        'success':function(response) { // JSONP 的回调函数
            alert(response.data.translations[0].translatedText); // 显示翻译结果
                }
        });
}
</script>
<input type="text" id="sourceText" />
<div onclick="doTranslate()">翻译</div>
```

Google Translate API 也提供了用于 JavaScript 语言的版本。代码清单 20.3 是其使用示例。它在内部使用的仍然是 JSONP，所以与代码清单 20.1 和代码清单 20.2 相比，其实质仅仅是隐藏了对 google.language.translate 的函数调用而已。

代码清单 20.3　使用了 Google Translate JavaScript API 的代码示例

```
<script src="https://www.google.com/jsapi?key=_YOUR_APIKEY_"></script>
<script type="text/javascript">
google.load("language", "1");

function doTranslate() {
    var text = document.getElementById("sourceText").value;

    google.language.translate(text, 'en', 'zh-CN',
        function(result) {
            if (result.translation) {
                alert(result.translation);     // 显示翻译结果
            }
        });
}
</script>
<input type="text" id="sourceText" />
<div onclick="doTranslate()">翻译</div>
```

20.2.4　微件（Google Translate Element）

本节最后介绍一下 Google Translate Element。根据 Google 的定义，Google Translate Element 提供的是一种名为 Web Elements 的框架。可以通过下面的站点了解更多有关 Web Elements 的信息。本书认为 Web Elements 也是一种 Web API。

http://www.google.com/webelements/[①]

我们可以在下面的页面中，以会话的形式设定各种选项，并获得 HTML 代码片段。

① Web Elements 在 2009 年推出，现在已经进行了一些调整。目前可以在 http://www.google.com/ig/directory 页面中找到一些范例，但不排除今后进一步调整的可能。这里仅作为对 Web API 的一种补充，做一下简单的介绍。——译者注

http://translate.google.com/translate_tools[1]

通过该页面获得的代码片段如代码清单 20.4 所示。将该代码片段粘贴至 HTML 中，就能够获得图 20.3 中的效果。

代码清单 20.4　Google Translate Element 的使用示例

```
This is a cat.
<div id="google_translate_element" style="display:block"></div>
<script>
function googleTranslateElementInit() {
    new google.translate.TranslateElement({pageLanguage: "en", includedLanguages: 'zh-CN'},
    "google_translate_element");
};
</script>
<script src="http://translate.google.com/translate_a/element.js?cb=googleTranslate
ElementInit"></script>
```

图 20.3　Google Translate Element 的显示效果



Google Translate Element 相当于上一章表 19.1 所说的插件式 API。正如本节所述，Google Translate API 包含 HTTP API、语言 API 及插件 API 这三种类型。这是一种典型的 Web API 演进模式。

20.3 Google Maps API

Google Maps 是一种具有代表性的 Web 应用，也是一种典型的 AJAX 式混搭应用[2]。Google 提供的这一 Web API 可以在各种 Web 站点及 Web 应用中被使用。

本章将会介绍 Google JavaScript Maps API（以下简称为 Google Maps API）。在说明 Google Maps API 之前，本节将先介绍以下两个概念。之所以要特地对其进行说明，是因为如果不知道有这两个功能，就很可能会使用 Google Maps API 对已有的功能重复开发。

- Google Static Maps API
- 我的地图

20.3.1　Google Static Maps API

在使用 Google Static Maps API 时，需要遵循指定的地图图像 URL 规则。通过 URL 查询参数，就能指定位置及尺寸。把下面这样的 HTML 代码片段插入 HTML 中，就能够在界面上显示地图。其结果如图 20.4 所示。

① 该 URL 已更新为 https://translate.google.com/manager/website/。——译者注

② Web 2.0 中的混搭指的是整合多个来源的信息或 Web API，并提供一体化服务的做法。——译者注

```
<img src="http://maps.google.com/maps/api/staticmap?center=Tokyo&size=512*512&sensor=false" />
```

图 20.4　Google Static Maps API 的结果



上面只是一个硬编码的 img 标签，完全没有自由度。不过，通过 JavaScript 来创建 img 标签就能实现地图的动态生成。img 标签和 script 标签一样，没有跨源限制，因此，这种方式也可以正常运行。

代码清单 20.5 是一个通过 Google Static Maps API 创建地图的示例，它将先通过 Geolocation API 获取当前位置，然后以该位置为中心，创建地图。

代码清单 20.5　动态创建 Google Static Maps API 所需的 URL

```
<script type="text/javascript">
    navigator.geolocation.getCurrentPosition(function(pos) {
        var lat = pos.coords.latitude;
        var lng = pos.coords.longitude;
        var img = document.createElement('img');
        img.src = 'http://maps.google.com/maps/api/staticmap?center=' + lat + ','
+ lng + '&zoom=14&size=512*512&sensor=true';
        document.body.appendChild(img);
    }
</script>
```

Google Static Maps API 显示的仅是 img 标签形式的地图，无法使用 Google Maps 所特有的地图拖放等操作。虽然它的局限性很大，不过反过来也限制了用户对地图进行操作，所以也有其适用之处。

20.3.2　我的地图

如图 20.5 所示，我的地图是一种能够在地图上添加标记及图形的交互式 Web 应用。我的地图在创建后将会生成唯一的 URL，可以在任意 HTML 的 iframe 中对其进行引用。

我的地图所具有的功能，比表面上所显示出来的要更为丰富，且仍然在不断改进。借助 Google Maps API 辛苦开发出的 Web 应用，有可能只是我的地图的一个粗糙的仿制品，因此，建议大家在开发前，仔细确认下所设想的功能是否是已经被实现了的。

图 20.5 我的地图



20.3.3 Google Maps API 的概要

按照前一章表 19.1 的分类方式，Google Maps API 仅提供了 JavaScript 语言的 API，而没有提供提供 HTTP API。不过众所周知，在语言 API 的内部使用的仍是 HTTP 通信（也就是说，其 URL 及查询参数没有在 API 级别上公开）。

由于 HTTP 通信本身不是加密的，因此可以对通信进行分析以模拟 API 的功能，不过采用这种做法的话，就倒退至了 Web 抓取的时代。直接使用语言 API 即可。

关于 API 的文档，可以参见以下站点。本书将介绍 Google Maps API 的第 3 版。

http://code.google.com/apis/maps/documentation/javascript/

在使用 Google Maps API 时，我们需要先了解一些术语的含义。表 20.3 对它们进行了总结。

表 20.3 与地图相关的术语

英语	中文
latitude	纬度
longitude	经度

20.3.4 简单的 Google Maps API 示例

本节将介绍如何在代码中使用 Google Maps API，以在浏览器中显示地图。代码清单 20.6 中的 JavaScript 代码，其实是一条通过 new 表达式，创建一个 google.maps.Map 对象实例。在其他几行中，仅仅构造了 new 表达式所需的参数而已。此处，new 表达式的结果被赋值给了 map，不过这不是必须步骤。

在代码清单 20.6 中，用于绘制地图的 JavaScript 代码和所要进行绘制的 HTML 元素组合在了一起。在代码中，是通过 HTML 元素的 id 属性将其结合的。此时，在 JavaScript 代码中引用了 id 的值为 map_canvas 的 div 元素。从代码中我们也能看出，id 属性的值是没有固定标准的，根据开发者的喜好选取即可。

代码清单 20.6 使用 Google Maps API

```
<body onload="initialize()">
<script src="http://maps.google.com/maps/api/js?sensor=false"></script>
<script type="text/javascript">
```

```
    function initialize() {
        var latlng = new google.maps.LatLng(35.6642722, 139.7291455);
        var myOptions = {
            zoom: 8,
            center: latlng,
            mayTypeId: google.maps.MapTypeId.ROADMAP
        };
        var map = new google.maps.Map(document.getElementById("map_canvas"), mpOptions);
</script>
<div id="map_canvas" style="width:100%; height:100%"></div>
</body>
```

只要通过代码清单 20.6 这样的代码，就能够在 HTML 内创建 iframe 并显示 Google Maps 界面，实现与其几乎相同的功能。在自己的代码中使用 Google Maps API，就能实现以下这些扩展功能。

- 捕获事件
- 显示控件、标记、HTML 元素（DOM 元素）等内容

这样一来，就能通过 JavaScript 实现地图的拖动、地图标记的添加，或者在用户对地图进行操作时执行一些自定义的处理。

我们来看一个通过代码对地图进行操作的例子。像下面这样，对 google.maps.MAP 实例的方法进行调用之后，就能够获取地图的状态，并对地图进行操作。

```
var map = new google.maps.Map(document.getElementById("map_canvas"), myOptions);
map.panBy(100, 100);
```

panBy 方法的功能是移动地图的显示区域（类似于相机的镜头摇摆效果）。

20.3.5　事件

基于 Google Maps 的程序设计的基本流程是，为对象设置事件处理程序，并根据事件设计相应的处理操作。也就是所谓的事件驱动型的代码。如果有在视窗系统中开发 GUI 程序的经验，对这种程序设计模型应该会很熟悉。

从代码清单 20.6 中我们可以看到，Google Maps API 将通过 new 来创建对象，是一种基于类的 API。姑且不论优劣，Google Maps API 是一种传统的基于类的事件驱动型 API。

Google Maps 的对象支持以下这些事件。

- click
- dblclick
- mouseup
- mousedown
- mouseover
- mouseout

这些事件的名称和功能，与第 3 部分介绍的 DOM 的事件相似。不过，这些事件是在 Google Maps API 层中进行定义的，并不是 DOM 中的事件。事件处理的基本形式为通过 addListener 方法将事件与事件处理程序相关联，如以下代码所示。

```
google.maps.event.addListener(目标对象， 表示事件名称的字符串， 表示事件处理程序的函数);
```

根据目标对象的不同，能用于表示事件名称的字符串也有所不同。关于各种对象所支持事件名称，请参见 API 的参考文档。代码清单 20.7 是一个事件处理的具体示例。

代码清单 20.7 Google Maps API 中事件处理的基本流程

```
<body onload="initialize()">
<script src="http://maps.google.com/maps/api/js?sensor=false"></script>
<script type="text/javascript">
    function initialize() {
         var latlng = new google.maps.LatLng(35.6642722, 139.7291455);
         var myOptions = {
              zoom: 8,
              center: latlng,
              mayTypeId: google.maps.MapTypeId.ROADMAP
         };
         // 创建 map 对象
         var map = new google.maps.Map(document.getElementById("map_canvas"), mpOptions);

         // 为 map 对象的 click 事件添加事件处理程序
         google.maps.event.addListener(map, 'click', function(event) {
              // 创建 marker 对象
              var marker = new google.maps.Marker({
                   position: event.laglng,
                     map: map
                     });

              // 为 marker 对象的 click 事件添加事件处理程序
              google.maps.event.addListener(marker, 'click', function(event) {
                   marker.setMap(null);
                 });
         })
    }
</script>
<div id="map_canvas" style="width:100%; height:100%"></div>
</body>
```

接下来，我们来介绍一下对代码清单 20.7 的执行方式。代码中直至通过 new 表达式创建 google.maps.Map 实例为止的部分，与代码清单 20.6 是相同的（下文中将会把 google.maps.Map 实例称为 map 对象）。之后，通过调用 addListener 方法，为 map 对象的 click 事件设置了事件处理程序。在代码清单 20.7 中，事件处理程序被指定为了一个匿名函数。当然，声明一个具有名称的函数并进行设定，也是没有问题的。

在事件处理程序中含有两个处理操作。其一是创建 google.maps.Marker 实例。在创建了 google.maps.Marker 实例之后，就能够在地图上显示标记（图标）。虽然也可以将实例的创建及显示分离，不过在代码清单 20.7 中，在创建实例时同时传递了 map 对象，实例的显示与创建是同时进行的。可以通过被传递至事件处理程序的参数，来获得点击的坐标（event.lat），并在该位置显示标记。

之后，还要为标记的 click 事件设置事件处理程序，并在事件处理程序内对标记对象调用 setMap 方法。如果 setMap 方法的参数被指定为了 null，则不会显示该标记，因此可以通过这种方式来隐藏点击位置的标记。

代码清单 20.7 的代码运用到了闭包。在 map 对象的事件处理程序中访问 map 变量，以及在标记的实际处理程序中访问 marker 变量，都是借助闭包实现的 [①]。

作为对比，代码清单 20.8 是一种基于类的事件处理方式。不同于基于闭包的事件处理，这种方式所写的代码更符合 JavaScript 的风格。不过，只要根据自己的喜好，自由地选择一种方式即可。代码清单 20.8 没有使用特殊的库，而是直接对 prototype 对象进行操作。在一些库中，隐藏了基于类的实现的具体细节，如果使用了这些库，则应注意需要遵循其相关规范。此外，代码清单 20.8 还使用了 ECMAScript 第 5 版中的 bind 方法。

① 关于闭包与 bind 方法，请参见第 6 章。

代码清单 20.8　基于类的事件处理风格

```
<body onload="initialize()">
<script src="http://maps.google.com/maps/api/js?sensor=false"></script>
<script type="text/javascript">
// MyEventListener 类的构造函数
function MyEventListener(map, latLng) {
    this.map = map;
    // 为 map 对象的 click 事件添加事件处理程序
    google.maps.event.addListener(map, 'click', this.show_marker.bind(this));
}

// MyEventListener 类的方法
MyEventListener.prototype.show_marker = function(event) {
    var marker = new google.maps.Marker({
        position: event.latLng,
        map: this.map
      });
    google.maps.event.addListener(marker, 'click', this.hide_marker.bind(this, marker));
}

// MyEventListener 类的方法
MyEventListener.prototype.hide_marker = function(marker, event) {
    marker.setMap(null);
}

function initialize() {
    var latlng = new google.maps.LatLng(35.6642722, 139.7291455);
    var myOptions = {
        zoom: 8,
        center: latlng,
        mayTypeId: google.maps.MapTypeId.ROADMAP
    };
    var map = new google.maps.Map(document.getElementById("map_canvas"), myOptions);
    new MyEventListener(map);
}
</script>
<div id="map_canvas" style="width:100%; height:100%"></div>
</body>
```

Google Maps API 还能捕获 DOM 的事件。例如，可以下面的代码这样，将 body 标签中的 onload 属性替换为 load 事件。

```
google.maps.event.addDomListener(window, 'load', initialize);
```

20.3.6　Geolocation API 与 Geocoding API

本节来介绍一种 Google Maps API 的应用实例，在该例中，结合使用了 Geolocation API 与 Geocoding API。Geolocation API 的功能是获取设备的位置信息。例如，对于支持 GPS 的设备，如果其浏览器支持 Geolocation，就能够通过 GPS 获取设备的位置信息。GPS 只是获取位置信息的方式之一，Geolocation 是否会使用 GPS 来获取位置信息，取决于具体的实现。如果 GPS 不可用，Google Maps API 中的 Geolocation 则会通过 IP 地址来推测设备的位置[①]。

关于 Geolocation API 的标准，请参见下面的站点。所发送的位置信息是一种隐私内容，因此，通常浏览器在调用 Geolocation API 时，都会向用户请求运行许可。如果用户不允许执行，调用将以失败告终，对此请加以注意。

http://www.w3.org/TR/geolocation-API/

Geocoding 是一种可以通过位置信息（经度及纬度）来查找住址，或反过来通过住址查找位置信息的

① 不过，通过 IP 地址获取的位置信息，其精度只能说是聊胜于无。

Web 服务。除了 Google，也有一些其他的 Web 服务通过 Web API 提供了 Geocoding 功能。这并非是一项 Google 专有的服务。

在代码清单 20.9 中，在 run 函数内通过 Geolocation API 获取了设备的当前位置（调用了 navigator.geolocation.getCurrentPosition）。如果调用成功，则会创建一个 google.maps.Geocoder 实例，并借助 Geocoding 来取得该位置信息所对应的住址。

代码清单 20.9 所得到的结果住址将通过 alert 显示，然后以当前位置为中心显示地图。

代码清单 20.9 在代码中使用 Geolocation 与 Geocoding 的示例

```
<body>
<script src="http://maps.google.com/maps/api/js?sensor=true"></script>
<script type="text/javascript">
function run() {
    if (navigator.geolocation && navigator.geolocation.getCurrentPosition) {
        // 调用 Geolocation API
        navigator.geolocation.getCurrentPosition(function(pos) {
            // Geolocation API 的回调函数
            var lat = pos.coords.latitude;
            var lng = pos.coords.longitude;

            // 调用 Geocoding API
            var geocoder = new google.maps.Geocoder();
            geocoder.geocode({ 'latLng': new google.maps.LatLng(lat, lng) },
                // Geocoding API 的回调函数
                function(results, status) {
                    if (status == google.maps.GeocoderStatus.OK) {
                        if (results[1]) {
                            alert(results[1].formatted_address);
                        }
                } else {
                    alert("Geocode error: " + status);
                }
            });

            var latlng = new google.maps.LatLng(lat, lng);
            var myOptions = {
                zoom: 14,
                center: latlng,
                mapTypeId: google.maps.MapTypeId.ROADMAP
            };
            new google.maps.Map(document.getElementById("map_canvas"), myOptions);
        });
    }
}
</script>
<div onclick="run()">获取当前位置</div>
<div id="map_canvas" style="width:100%; height:100%"></div>
</body>
```

20.4 Yahoo! Flickr

Flickr 是一个典型的照片分享服务，在 Web 2.0 这个词刚出现时就已经存在。现在，Flickr 已经被 Yahoo! 收购。可以通过下面的 URL 查看其 Web API 的参考文档。

http://developer.yahoo.com/flickr/

http://www.flickr.com/services/api/

根据前一章中表 19.1 的分类，Flickr 的 Web API 中仅提供了 HTTP API，而 HTTP API 又分为 REST、XML-RPC 及 SOAP 三种类型。其语言 API 是由第三方提供的。由于这是一种流行的 Web API，因此很

多程序设计语言都已经有了相对应的语言 API。

Flickr 的 Web API 被公开时，RESTful 这一概念还没有像现在这样流行。这时，对于 RESTful 的 URL 设计，还没有取得完全一致的意见，将非 SOAP 的 Web API 称为 RESTful API 的做法也才刚出现。因此，尽管 Flicker 的 REST API 在形式上确实不是 SOAP 类型，但其设计的核心思想却是 RPC。虽然它被称为 REST 类型，但却不是 REST 风格的面向资源（面向文档）的设计方式。与之后将要说明的 Twitter API 比较之后，大家就能很容易地理解这一点。以现在的标准来看，Flickr 的 API 设计有些陈旧。当然，说一个设计是新还是旧是比较主观的，大家只要将这一观点作为一种参考即可。

Flickr 的 Web API 将会把请求发送至以下的 URL。用 Flickr API 的术语来说，该 URL 被称为端点（endpoint）。

http://secure.flickr.com/services

http://api.flickr.com/services

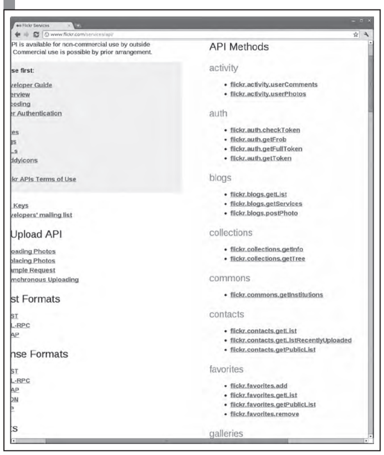
可以通过发送至端点 URL 的请求参数，来对各种操作进行设定。其中最为重要的是 method 参数。method 参数的值所指定的是操作名称。下面是一个请求 URL 的例子，在下一节中还会进行详细说明。

http://api.flickr.com/services/rest/?method=flickr.test.echo

该 URL 将会调用 flickr.test.echo 方法。在 API 参考文档中，记载了这类方法名的一览表。通过 Web API 可以实现照片的获取、更新及添加注释等操作。虽然前面讲到 Flickr API 的设计有些陈旧，但它毕竟已有一定的历史，功能很全面。

图 20.6 是打开 API 参考文档时的界面截图。其中，flickr.activity.userComments 等相当于方法名称。

图 20.6　Flickr 的 Web API 方法一览



每种方法都具有其所需的参数及选项参数。这些参数通过 URL 的查询参数相连。这就是 Flickr 的 Web API 的基本结构。

20.4.1　Flickr Web API 的使用

要使用 Flickr 的 Web API，必须有其 API 密钥。而要获取 API 密钥，则必须有 Yahoo! ID。这些都是可以免费获取的。访问以下页面中的 Get an API Key 链接，就可以获得 API 密钥。

http://www.flickr.com/services/

Flickr API 对 JSONP 提供了支持。因此，可以在客户端 JavaScript 中使用该 API。不过，在通过客户端 JavaScript 使用该 API 时有一个很大的问题，即 API 密钥无法加密。由于 API 密钥将会被传递给 HTTP 请求的查询参数，因此无法将其隐藏。

不过，API 密钥是否需要加密取决于开发者的想法。考虑一下 API 密钥被盗后可能产生的实际损失，会发现情况有些微妙。即使获取了 API 密钥，也无法盗取相应的 Flickr 账号。对此人们的观点各不相同，不可一概而论。本书假定不需要对 Flickr 的 API 密钥进行加密，并以此为前提在客户端 JavaScript 中调用该 Web API。

和 Google Translate API 一样，首先通过 curl 指令来确认其功能（图 20.7）。

图 20.7 通过 curl 对功能进行确认

```
$ export APIKEY= 所获取的 API 密钥

# echo 仅会返回所传递的 name 参数的值（这对于早期的功能确认来说，是一种很方便的方式）
# 默认的响应输出格式是 XML
$ curl "http://api.flickr.com/services/rest/?method=flickr.test.echo&name=hello&api_key=${APIKEY}"
<?xml version="1.0" encoding="utf-8" ?>
<rsp stat="ok">
<method>flickr.test.echo</method>
<name>hello</name>
<api_key> 所获取的 API 密钥 </api_key>
</rsp>

# 将 format 参数指定为 json 之后，就会返回 JSONP 格式的响应
# JSONP 中函数名的默认值为 jsonFlickrApi
$ curl "http://api.flickr.com/services/rest/?method=flickr.test.echo&name=hello&api_
key=${APIKEY}&format=json"
jsonFlickrApi({"method":{"_content":"flickr.test.echo"}, "name":{"_content":"hello"},
"api_key":{"_content":" 所获取的 API 密钥 "}, "format":{"_content":"json"}, "stat":"of"})

# 可以将 jsoncallback 参数指定为 JSONP 的函数名
$ curl "http://api.flickr.com/services/rest/?method=flickr.test.echo&name=hello&api_
key=${APIKEY}&format=json&jsoncallback=myfunc"
myfunc({"method":{"_content":"flickr.test.echo"}, "name":{"_content":"hello"},
"api_key":{"_content":" 所获取的 API 密钥 "}, "format":{"_content":"json"}, "stat":"of"})

# 搜索图片（搜索字符串为 gozilla）
$ curl "http://api.flickr.com/services/rest/?method=flickr.photos.search&text=gozilla
&api_key=${APIKEY}&format=json&jsoncallback=myfunc"
myfunc({"photos": 下略 })
```

20.4.2 Flickr Web API 的使用实例

下面的代码将会通过 Flickr API，以关键字搜索照片，并将结果显示出来（代码清单 20.10）。其中 _YOUR_APIKEY_ 需要替换为所获取的 API 密钥。

下面，我们来说明一下代码清单 20.10 中的要点。jQuery 隐藏了 JSONP 的调用。将 $.ajax 函数的参数对象的 dataType 属性设定为 jsonp 的话，就能隐藏内部的 JSONP 处理（诸如动态创建 script 标签等操作）。默认情况下，jQuery 会将用于指定 JSONP 回调函数的 HTTP 查询参数名假定为 callback。而 Flickr 的查询参数名是 jsoncallback，因此，必须在 jQuery 中显式地对其（jsonp 属性的值）进行修改。

关于响应的格式，请参见 Flickr 的参考文档。在枚举响应时，使用的是 ECMAScript 第 5 版中的 forEach，对此请加以注意。搜索操作的响应格式如下，根据该标准可以获得图片的 URL。

http://farm{$farm-id}.static.flickr.com/{$server-id}/{$id}_{$secret}.jpg

代码清单 20.10 Flickr Web API 的使用示例

```
<body>
```

```
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.6.1/jquery.min.js"
type="text/javascript"></script>
<script>
function doSearch() {
    $.ajax({
        'type':'GET',
        'url':'http://api.flickr.com/services/rest/',
        'data': {method:'flickr.photos/search', text:$('#queryString').val(), api_key:'_
            YOUR_APIKEY_', format:'json'},
        'dataType':'jsonp',
        'jsonp':'jsoncallback',
        'success':function(data) {
            var content = $('#content');
            if (data.photos && data.photo) {
                data.photos.photo.forEach(function(photo) {
                    var img_url = 'http://farm' + photo.farm + '.static.flickr.com/'
                        + photo.server + '/' + photo.id + '_' + photo.secret + '.jpg';
                    content.append('<img src="' + img_url + '"</img>');
                });
            }
        }
    });
}
</script>

<input type="text" id="queryString" name="queryString" />
<div onclick="doSearch()">Go</div>
<div id="content"></div>
</body>
```

在 Flickr 的 Web API 中，有一些是需要进行验证的。Flickr 自定义了一套验证机制，它类似于前一章介绍的 OAuth。从历史的角度来看，Flickr 是使用 OAuth 这样的权限转让协议的先驱，OAuth 是其标准化之后的产物。不过，这仅限于图 19.5 的情况。在 Flickr API 中，没有类似于图 19.6 中的 OAuth 用户代理流程的机制。因此在 Flickr 中，需要进行验证的 Web API 是无法在客户端 JavaScript 中进行调用的。

20.5 Twitter

与之后将要介绍的 Facebook 一样，Twitter 在公开了 Web API，使第三方应用大量涌现并构筑了其软件生态环境之后，获得了巨大的成功[①]。对 Twitter 这一服务，在此不再说明。从 Web API 的角度来看，Twitter 所提供的服务较为单一。通过对所谓的 Tweet（推文）进行更新与获取，以及以关注的形式连接用户，Twitter 构造出了一种社交图谱。这就是其 Web API 的主要操作对象。

关于 Twitter API 的参考文档，请参见下面的 URL。

http://dev.twitter.com/

Twitter API 可以分为 REST API 与 Streaming API 两类。本书介绍的是 REST API。在 REST API 中，搜索 API 的规则略有些不同。这仅仅是由于历史原因所致，并非技术问题。Streaming API 无法在客户端 JavaScript 中使用，因此我们将省略这部分说明。

20.5.1　搜索 API

本节将先介绍一下 Twitter API 中简单而实用的搜索 API。搜索 API 将会返回包含参数所指定的搜索

① Twitter 是当前最流行的微博客服务，在翻译本书时全球已有超过 2 亿用户，在大多数国家处于绝对领先地位。国内的新浪微博等是模仿 Twitter 并增加了一系列本土化功能的服务。——译者注

字符串的 Tweet。Twitter 服务本质上并不要求精确的搜索功能，对于这一搜索功能，只要将其理解为，在已经存在的推文（Tweet）中找出一些与搜索相匹配的内容即可。

将搜索 API 的请求 URL 格式中可以改变的部分写为 {parameter} 之后，就得到了下面这样的请求 URL。

```
http://search.twitter.com/search.{format}?={ 搜索字符串 }&{ 其他参数 }
```

format 部分可以被替换为 json 或 atom 等字符串。例如，对于搜索字符串为 emacs，响应格式为 JSON 的请求 URL，下面是一种最简单的写法。

http://search.twitter.com/search.json?q=emacs

关于其他参数，请参见 API 的参考文档。可以像下面这样，通过 curl 指令来确认其功能。接下来，大家亲自确认一下搜索的结果吧。

```
# 参数的说明
# q=emacs 搜索关键字（必须）
# locale=ja 日语
# geocode=35.6642722,139.7291455,10km 加入位置信息
$ curl "http://search.twitter.com/search.json?q=emacs&locale=ja&geocode=35.6642722,
139.7291455,10km"
```

如果只需在服务器端使用 Twitter API，所有的说明就已经完成了。具体的实现方式则应遵循各个程序设计语言的习惯。既可以在代码中直接使用 HTTP，也可以使用库来隐藏 HTTP 通信，这都是开发者的自由。Twitter 并没有提供语言 API，不过已经有了很多的第三方语言 API。主流的程序设计语言通常都有其对应版本。

而另一方面，对于客户端 JavaScript，一如既往地存在跨源限制。通过使用 JSONP，就可以在 Twitter API 中回避跨源限制。像下面这样传递了 callback 参数之后，就能返回 JSONP 形式的响应。

```
$ curl "http://search.twitter.com/search.json?q=javascript&locale=ja&geocode=35.66
42722,139.7291455,10km&callback=myfunc"
myfunc({"results":[{... 省略 }], ... 省略 })
```

通过客户端 JavaScript 调用该 JSONP 是很容易的，因此就不再举例[①]。

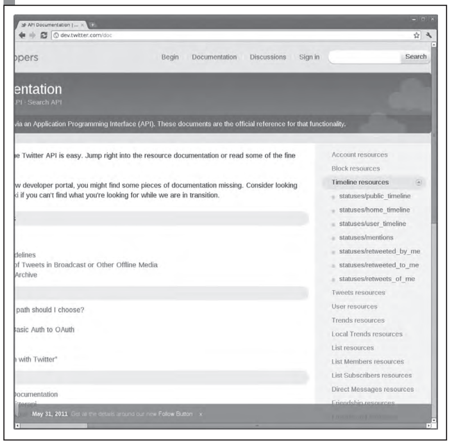
20.5.2　REST API

接下来，我们说明一下 Twitter API 中的 REST API。关于 REST API 的一览表，请参见站点的参考文档。REST API 具有多种功能，通过该 API，可以执行对 Tweet（推文）的显示、更新，及通知用户与关注用户等操作，以取得有这些操作所构筑的社交图谱。还可以通过 API 获得 Tweet 趋势等相关信息。尽管下面的说法中，有些鸡生蛋蛋生鸡的成分，不过不难理解，Twitter 这一服务能够成功的主要原因之一正是由于它提供了丰富的 API。

图 20.8 是打开了 Timeline resource 的 API 参考文档时的界面。值得注意的是，API 参考文档也用到了资源这一术语。

① 可以参见 Google Translate API 及 Flickr 相关的小节，其中有类似代码。搜索 API 不需要进行用户验证。

图 20.8　Twitter REST API



有很多的 Web API 虽然自称是 RESTful，但却没有采用 RESTful 的 URL 设计方式。而 Twitter 的 REST API 是基于正确的 RESTful 思想设计的。也就是说，它不是 RPC 形式的 API，而是面向资源的 Web API。从 URL 设计中，就能体现出其面向资源的特征。与 Flickr API 进行比较的话，就能很容易地发现这一点。对于 Flickr API 的 URL，传递给 method 参数的是方法名称（操作名称）。与此同时，Twitter API 的 URL 的操作对象是资源名称。其操作是通过 HTTP 的 GET 及 POST 方法来指定的。

将 REST API 的请求 URL 格式中可以改变的部分写为 {parameter} 之后，就得到了下面这样的请求 URL。

http://api.twitter.com/{version}/{resource}.{format}?{ 参数 }

在执笔本书时，version 仅有 1 这一种。format 则可以指定为 json、xml、rss 及 atom 这几个字符串的其中之一。resource 部分所写的是资源名。例如对于资源为 statused/public_timeline，响应格式为 JSON 的请求 URL，下面是一种最简单的写法。

http://api.twitter.com/1/statuses/public_timeline.json

关于 REST API 所支持的资源名，请参见 API 的参考文档。原则上，请求 URL 都是基于 http://api.twitter.com 扩展而成的。

Twitter 的 REST API 对 JSONP 提供了支持。因此，可以在客户端 JavaScript 中使用该 API，且不会发生跨源限制。像下面这样将 callback 参数传递给请求 URL 之后，就能获取 JSONP 格式的响应。

http://api.twitter.com/1/statuses/public_timeline.json?callback=myfunc

通过客户端 JavaScript 调用该 JSONP 是很容易的，所以不再举例。大家可以参见 Google Translate API 及 Flickr 相关的小节，其中有类似代码。之所以不介绍实际的例子还有一个原因。比起直接通过客户端 JavaScript 调用 Twitter API，更推荐大家使用下面的 @anywhere 或 Twitter Widget。

20.5.3　Twitter JS API @anywhere

有一些 Twitter API 是需要进行验证的。例如，执行 Tweet 发送功能的 API 便是如此。通过前一章介绍的 OAuth 2.0 方式，Twitter API 已经解决了这一问题。如果要在服务器端调用 Twitter API，可以通过图 19.5 那样的流程，让用户进行权限转让，然后调用 Web API。可以自己来实现这一流程。不过如果使用的是流行的程序设计语言，则会有相应的库。使用这些库来实现权限转让会更加容易。

同样地，如果要通过客户端 JavaScript 调用需要验证的 Web API，则可以通过 OAuth 2.0 的用户代理流程，来实现权限的转让（图 19.6）。有一种名为 @anywhere 的 API，可以隐藏这一 OAuth 用户代理流程。根据表 19.1 的分类，之前小节介绍的都是 HTTP API，而 @anywhere 则是针对客户端 JavaScript 的语言 API。

为了使用 @anywhere（正确地说是为了使用 OAuth），必须要取得 API 密钥。可以在下面的 URL 中注册应用，以获取 API 密钥（图 20.9）。

http://dev.twitter.com/anywhere/apps/new

图 20.9 注册应用

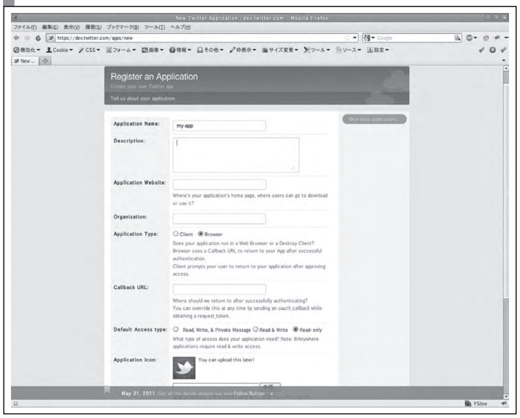


图 20.10 @anywhere



代码清单 20.11 是使用 @anywhere 的代码示例。_YOUR_APIKEY_ 部分需要替换为所获取的 API 密钥。代码中的 ' 任意的 Twitter 账户 ' 部分则需要替换为感兴趣的 Twitter 账户。如果是要把代码清单 20.11 的代码片段粘贴至自己的站点中，以让访问者关注自己，则可以将其替换为自己的 Twitter 账户。

代码清单 20.11 将会在浏览器界面中显示一个 follow me 的链接。在点击了链接之后，将会跳转至新的界面，选择是否要关注所显示的 Twitter 账户（图 20.10）。在图 20.10 中，pjs2011a 是应用名称。该界面正在向用户请求许可，以进行权限转让。在获得了授权之后，将会以用户所具有的权限来执行 Web API，进行关注。这里的用户指的不是显示该链接的 Web 站点的所有者，而是通过浏览器访问了该页面的用户。

代码清单 20.11 @anywhere 的代码示例

```
<script src="http://platform.twitter.com/anywhere.js?id=_YOUR_APIKEY_&v=1"></script>
<script type="text/javascript">
twttr.anywhere(function (T) {
    T('#follow').followButton(' 任意的 Twitter 账户 ');
});
</script>

<span id="follow" />
</body>
```

本节之后将对代码清单 20.11 中的要点进行说明。一旦完成了 anywhere.js 的读取，就会隐式地创建一个 twttr 对象。@anywhere 的基本使用方式是调用 twttr 对象的 anywhere 方法。在调用过程中，需要将一个含有处理操作的回调函数传递给这一方法的参数。在该方法的内部，对跨源限制进行了处理，同时执行了权限转让的操作。不过，开发者在实际使用时并不需要在意这些细节。

可以随时调用 twttr.anywhere 方法。在代码清单 20.11 中，该方法将会在 HTML 文件读取完成后被执行，也可以改为在点击时执行该方法。twttr.anywhere 方法的回调函数的参数所需的是一个 Twitter API

客户端对象。代码清单 20.11 通过变量 T 来接收该对象。下面的结论或许有些不合常规，不过，对该对象进行方法调用，就等于使用 @anywhere API。而代码清单 20.11 中的 '#follow'，将会通过 CSS 选择器样式来获取 DOM 元素。这种方式与 jQuery 的代码风格是一致的。这是因为，@anywhere 的内部使用的正是 jQuery。除了 followButton，Twitter API 客户端对象还提供了一些其他能够调用的方法，更为详细的信息，请参见 API 的参考文档。这些 API 非常简单，不必特地说明。这也是 Web API 不断改进的一种体现。

20.5.4　Twitter Widget

在表 19.1 中，Web API 的演进被分为了 HTTP API、语言 API 及插件 API。迄今为止，我们已经介绍了 HTTP API 和语言 API（@anywhere）。最后将会介绍 Twitter Widget 这一插件 API。Twitter Widget 本质上是一种自定义标签，它将在客户端被解释执行。只要在 HTML 中粘贴相应的代码片段，就能够隐藏所有的 JavaScript 代码（包括对跨源限制问题的回避，以及用户验证操作的处理）。

Twitter Widget 分为 JavaScript 版与 iframe 版[①]，这里介绍的是 JavaScript 版。只要在 HTML 中写下以下代码片段，就能够在浏览器界面中显示 Tweet Button 及 Tweet 数（图 20.11）。

```
<script src="http://platform.twitter.com/widgets.js" type="text/javascript"></script>
<a href="http://twitter.com/share" class="twitter-share-button">Tweet</a>
```

图 20.11　Tweet 按钮与 Tweet 数



20.6　Facebook

20.6.1　Facebook 应用的发展历程

对于 Facebook 这一服务，想必已无需再多做介绍。在执笔本书时，Facebook 是发展势头最好的 SNS（社会化网络服务，Social Network Service）。Facebook 如此有名，几乎已经成为了 SNS 这一类型的代名词[②]。使 Facebook 一举成名的是 Facebook 所提供的各种应用。Facebook 应用其实有一段不为人知的发展历程，在此我们对其稍做总结。

最早的 Facebook 应用分为两种形式。我们可以通过在（托管了 Facebook 应用的第三方）服务器端调用 REST API 的形式，或者以使用 FBML 这一自定义标记语言的形式，来实现一个 Facebook 应用。FBML 是一种由 Facebook 服务器进行解释的自定义标记语言，这一技术与 PHP 及 JSP 处于同一层面。虽然从用户的角度来看，这两种形式的应用都是嵌于（诸如 iframe 等的）Facebook 页面中的，但其执行方式却有所不同。图 20.12 与图 20.13 分别是两种方式的执行原理。

① 更详细的信息，可以参见以下页面：https://twitter.com/about/resources/buttons。——译者注

② Facebook 是当前最为流行的 SNS 网站，在翻译本书时全球已有超过 11 亿用户，在大多数国家处于领先地位。国内的人人网等是模仿 Facebook 并增加了一系列本土化功能的服务。——译者注

图 20.12 早期的 Facebook 应用（iframe 版）

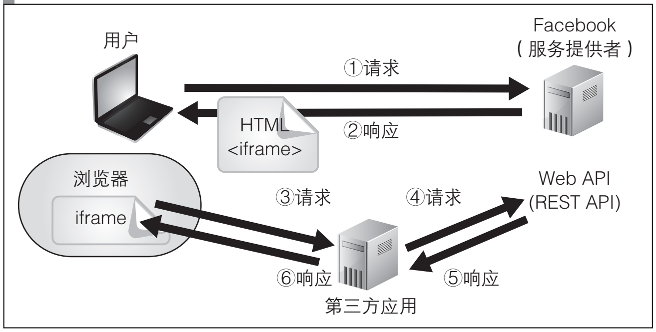
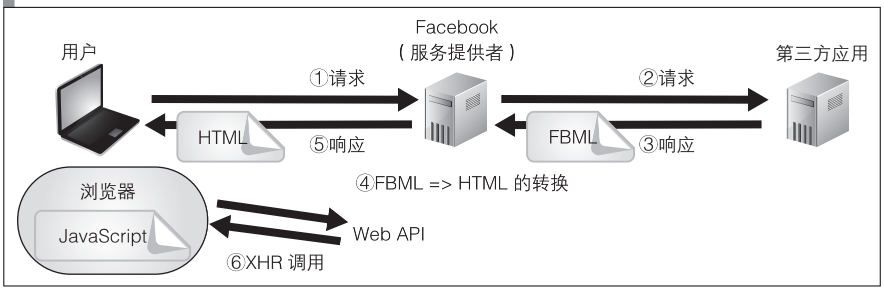


图 20.13 早期的 Facebook 应用（FBML 版）



Facebook 的 API 已经发生了巨大的进化。如今，Facebook 应用的执行方式可以分为以下三种。

- 在服务器端调用 Graph API（并通过 iframe 在 facebook 的容器内显示）
- 在客户端（浏览器）中使用 JavaScript API
- 在客户端中使用插件（Social Plugins）

以下是各种执行类型的图示（图 20.14、图 20.15）。通过插件的使用方式与使用 JavaScript API 时的流程相同，故在此省略。

图 20.14 Facebook 应用（在服务器端调用 Graph API）

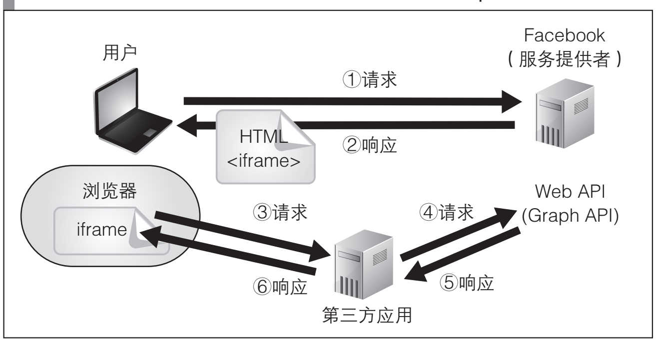
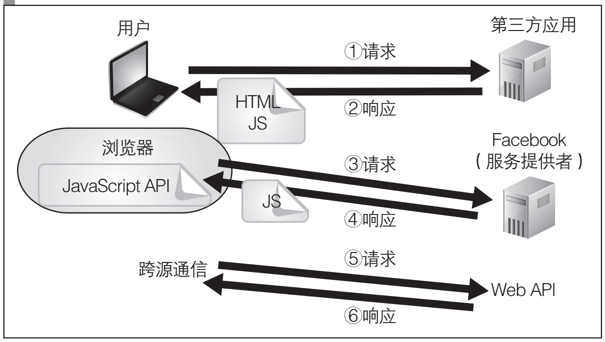


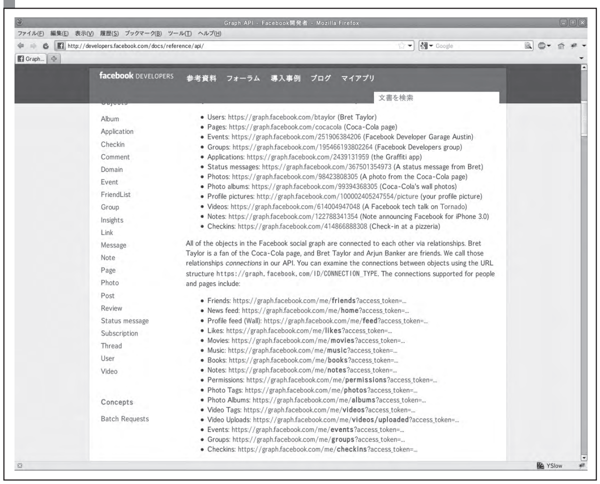
图 20.15　Facebook 应用（在客户端中使用 JavaScript API）



Graph API 是一种典型的 RESTful Web API。在调用了 Web API 之后，就能搜索 Facebook 用户，并取得用户之间的好友关系（即所谓的社交图谱），或者进行消息发送等[①]。

通过 Graph API，就能对 Facebook 的 Web API 所能提供的功能有一个整体把握。在执笔本书时，其功能仍在不断增加，给人以顶级 Web API 的印象。图 20.16 是打开 Graph API 的参考文档时的屏幕截图。

图 20.16　Facebook 的 Graph API



目前，Graph API 并不是以通过客户端 JavaScript 直接使用为前提设计的，需要通过 JavaScript API 来对其进行利用。在这时，还能同时使用 XFBML 这一 Facebook 自定义的标记语言，客户端 JavaScript 将会对其进行解释。而插件（Social Plugin）则能够隐藏 XFBML 与 JavaScript API，以进一步简化操作。

在 Facebook 的插件中，Like 按钮（“赞!”按钮）非常有名。用表 19.1 中的术语来说的话，这相当于一个插件 API。虽然与 JavaScript API 相比，插件 API 的自由度较低，不过只要以类似于博客插件的方式将代码片段插入 HTML 内，就能够实现相应的功能。在这一过程中无需关心内部的具体实现。

20.6.2　Facebook 的 JavaScript API

接下来要介绍的 JavaScript API，相当于在表 19.1 中所说的语言 API。大家可以通过下面的站点查看

① Graph API 的前身是 REST API，不过与名称相反，它并不是一个 RESTful 设计风格的 Web API。而 Graph API 则是一个 RESTful 设计风格的 Web API。

API 的参考资料。

http://developers.facebook.com/docs/reference/javascript/

JavaScript API 可以分为以下四大类别。

- Graph API 对应的包装 API（通过 FB.api 方法调用）
- XFBML 类 API（XFBML 是一种通过客户端 JavaScript 解释的标记语言）
- 事件处理程序类 API
- 其他

要使用 Facebook 的 JavaScript API，要先在 Facebook 中注册该应用，并获得应用 ID。在注册应用时会用到 Facebook 账户，如果还没有 Facebook 账户，则需要注册一个。账户的注册和应用的注册都是免费的。

可以访问以下 URL 以注册应用。

http://developers.facebook.com/setup/

在注册过程中，至少需要输入应用程序的名称及 URL（用于 OAuth 的回调函数）。之后便可以获得应用 ID 与密钥（用于图 19.5 中的 OAuth 流程）。如果所调用的 Web API 需要执行验证操作，则需要使用 OAuth 用户代理流程（图 19.6）。关于 OAuth 的详细内容，请参见之前的章节。

下面是一段简单的使用 JavaScript 的范例代码（代码清单 20.12）。其中 _YOUR_APPID_ 部分需要替换为所获取的应用 ID。

代码清单 20.12　Facebook JavaScript API 的使用示例

```
<div id="fb-root"></div>
<script src="http://connect.facebook.net/zh_CN/all.js"></script>
<script type="text/javascript">
FB.init({
    appId : '_YOUR_APPID_',
    status : true,
    cookie : true,
    xfbml : true
});

FB.login(function(response) {
    if (response.session) {
        FB.api('/me', function(response) {
            alert(response.name);
        });
    } else {
        ;   // 用户取消了登录
    }
});
</script>
```

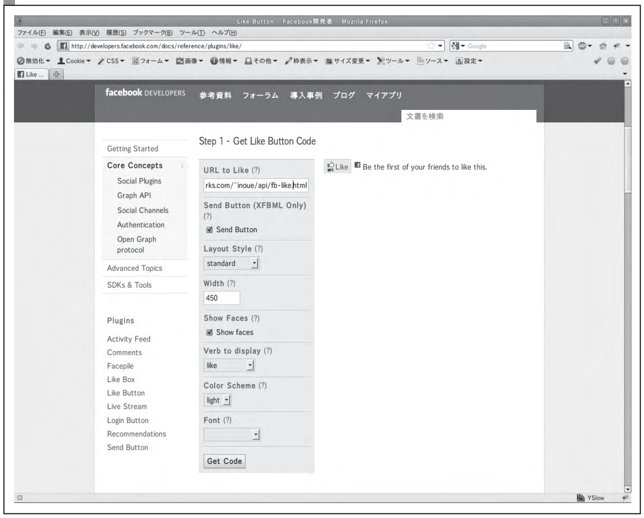
20.6.3 Facebook 的插件

根据表 19.1 的分类，Facebook 插件属于插件型 API。OAuth 用户代理流程所实现的权限转让操作，以及用于回避跨源限制的代码，都被隐藏于了 API 内部。个人认为，Facebook 提供了目前最先进的插件型 API 的 Web 服务。

Facebook 插件的使用非常简单。首先要做的是访问以下的 URL，以会话的形式获取所需的代码（图 20.17）。该插件分为 iframe 版与 XFBML 版，可以以会话的形式选择。

http://developers.facebook.com/docs/plugins/

图 20.17　Facebook 的 Like 按钮



只要将获得的代码粘贴至 HTML 内，就可以实现相应的功能。代码清单 20.13 的代码片段能够在浏览器界面中显示 Like 按钮。对于代码内的 _YOUR_URL_ 部分，需要改写为托管了该 HTML 的站点 URL。

代码清单 20.13　Like 按钮

```
iframe 版
<iframe src="http://www.facebook.com/plugins/like.php?href=_YOUR_URL_"
        scrolling="no" frameborder="0"
        style="border:none; width:450px; height:80px"></iframe>
XFBML 版
<script src="http://connect.facebook.net/zh_CN/all.js#xfbml=1"></script>
<fb:like></fb:like>
```

和上一节中的 Twitter Widget 一样，上述代码的结构非常简单，这里不再说明。

20.7　OpenSocial

为了对抗 Facebook，以 Google 为首的公司推出了 OpenSocial 这一开放式 Web API 标准。在执笔本书时，OpenSocial v2.0 的标准正在制订过程中。目前，OpenSocial 在技术上远落后于 Facebook API，仅处于追赶者的地位[①]。更为详细的信息，请参见以下 URL。

http://www.opensocial.org/

http://code.google.com/apis/opensocial/

OpenSocial 的 API 可以分为多种类型。按照本书的标准，可以将其分为两大类。

- RESTful API（包括 JSON-RPC API）
- JavaScript API（包括插件 API）

① 尽管在美国及日本，一些大型的网站已经开始引入了 OpenSocial，不过在国内，由于各种各样的原因，OpenSocial 恐怕无法得到普及。同时，国内的腾讯、新浪等公司也正在推广自己的 Web API。——译者注

RESTful API 相当于表 19.1 中的 HTTP API。它又包含了 RESTful 设计风格的 API 与基于 JSON-RPC 的 API 这两种类型。在标准中并没有规定必须对 JSONP 提供支持，是否要支持 JSONP 是对容器进行实现的开发者的自由。不过，下一节也会讲到，当前，OpenSocial 的目标运行架构并不会产生跨源限制问题，因此，即使不支持 JSONP，也不会有什么问题。

通过客户端 JavaScript 来使用 OpenSocial 时，通常会使用 JavaScript API 的方式。根据表 19.1 中的分类，这相当于语言 API。本节将只说明 OpenSocial 的 JavaScript API。

OpenSocial 的基本架构

OpenSocial 的早期原型是 iGoogle，因此，OpenSocial 的基本架构是基于容器与插件的。如图 20.18 与图 20.19 所示，它主要有两种主要的形式。可以将这两种形式分别命名为 iframe 型与代理型，它们分别对应于图 20.12 与图 20.13 所示的 Facebook 应用类型。

图 20.18 OpenSocial 的基本架构（iframe 型）

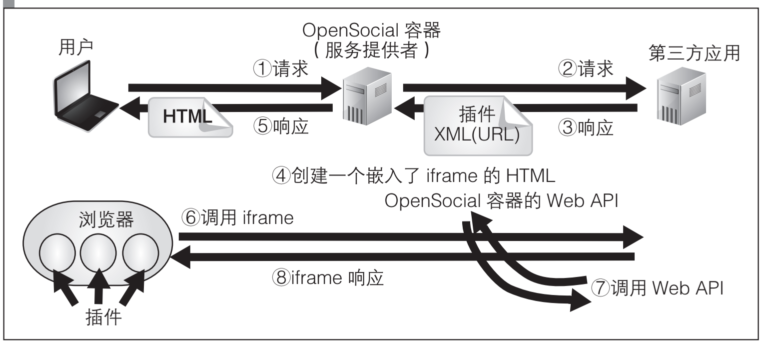
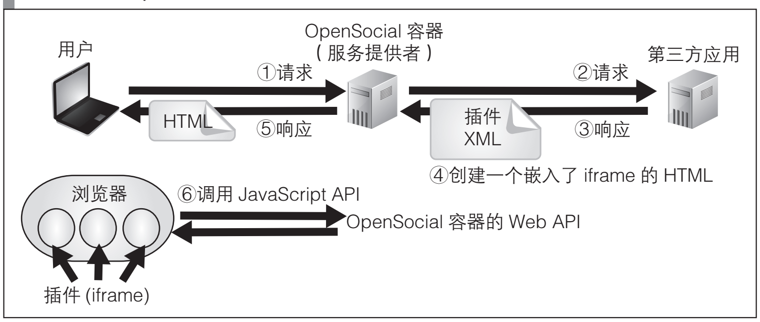


图 20.19 OpenSocial 的基本架构（代理型）



本书将不会介绍 iframe 型。这是因为，iframe 型的架构通常是用于服务器端的 OpenSocial 调用中的。代理型则将默认使用 JavaScript API。可以以任意的第三方服务器来托管该插件。从浏览器的角度来看，容器充当了中介者的角色，因此，所谓的客户端 JavaScript 跨源限制问题也就不存在了。

■ Google Friend Connect

为了与 Facebook Connect 竞争，出现了 Google Friend Connect 这一 API[①]。Facebook Connect 是一种基于 Social Plugin 的技术，前一节已经介绍过。它隐藏了 OAuth 用户代理流程以及跨源通信问题。这一语言 API 甚至还隐藏了插件 API。OpenSocial 也很可能会以此为方向发展，那样的话，其基本架构将会逐

① 最近，Google 为构建自己的 SNS 而进行了大量努力。更为详细的内容，请参见 Google+ 相关的参考文档。——译者注

渐从图 20.19 转变为图 20.15 的形式。

■ OpenSocial 的 API

如图 20.19 所示，在 OpenSocial 中，插件是在容器内显示的。插件本质上是一种通过 XML 描述的定义文件。代码清单 20.14 是其原型定义。

代码清单 20.14　OpenSocial 插件的原型定义

```
<?xml version="1.0" encoding="UTF-8"?>
<Module>
    <ModulePrefs title=" 插件的标题 ">     <!-- 可以将属性指定为作者信息或插件尺寸等内容 -->
        <Require feature="osapi"></Require>
        在这里声明所需的 OpenSocial feature
    </ModulePrefs>
    <Content type="html">
    <![CDATA[
    <style type="text/css"> 这里所写的是 CSS 代码 </script>
    <script type="text/javascript">
        这里所写的是 JavaScript 代码（可以使用 OpenSocial 的 JavaScript API）
    </script>
    这里所写的是 HTML 代码
    ]]>
    </Content>
</Module>
```

插件是由通常的 HTML（不过只需要写有 body 元素）、CSS 及 JavaScript 等元素所构成的[①]。容器将对通信进行中转，从 XML 中抽出 HTML 等内容，并向浏览器返回一个嵌入了这一 HTML 的 HTML 文件。广义的 OpenSocial 程序设计指的就是插件的定义文件的设计。狭义的 OpenSocial 程序设计则是指设计插件内的 JavaScript 代码。

这里的 JavaScript 代码将会在客户端执行，并能够使用 OpenSocial 的 JavaScript API。JavaScript API 包括多种类型，其中社交图谱类 API 用于和 OpenSocial 容器进行通信，UI 类 API 则用于操作插件的 UI。

■ Apache Shindig

Apache Shindig 是一种开源的 OpenSocial 容器。关于该容器的下载与安装方法，请参见下面的 URL。

http://shindig.apache.org/

代码清单 20.15 是显示了 Shindig 基本界面的 HTML 示例。该基本界面将会在指定的 URL 中寻找插件文件（my-gadget.xml），并创建一个嵌入了这一插件的界面。需要将这段 HTML 代码配置于运行有 Shindig 的服务器上，并在浏览器访问 Shindig 服务器时打开该页面。如此一来，界面中就会显示出由 my-gadget.xml 所描述的插件。

my-gadget.xml 是插件的定义文件，可以被配置于任意的 Web 服务器（第三方应用）上。它可以使用任意名称作为文件名。之后，OpenSocial 容器，也就是 Shindig 服务器将会获取该文件。之后将会说到，由于插件定义文件是由客户端 JavaScript 代码所写的，因此乍一看，很容易误以为它会被浏览器直接读取，对此请加以注意。

代码清单 20.15　通过 Shindig 在基本界面中显示插件

```
<html>
<head><title>Simple Container</title>
<script type="text/javascript" src="/gadgets/js/shindig-container:rpc.
js?c=1&debug=1&nocache=1"></script>
<script type="text/javascript">
```

① 只要通过代码 type="url" 将 Content 元素的 type 设定为 url，并将其 href 属性设定为相应的 URL，就能将其转换为一个插件，并在 iframe 内显示外部 HTML（图 20.18 的形式）。

```
function init() {
    var specUrl = '托管了 my-gadget.xml 的 URL';
    var gadget = shindig.container.createGadget({specUrl: specUrl});
    shindig.container.addGadget(gadget);
    shindig.container.layoutManager.setGadgetChromeIds(['my-gadget']);
    shindig.container.renderGadget(gadget);
}
</script>
</head>
<body onLoad="init()">
    <div id="my-gadget" class="gadgets-gadget-chrome"></div>
</body>
</html>
```

代码清单 20.16 是 my-gadget.xml 的最简形式。它仅仅是在代码清单 20.14 的原型中写上了 HTML 代码而已。将代码清单 20.15 中，my-gadget.xml 的 URL 设定为正确的值之后，就能在 Shindig 中显示该插件，并在界面中显示 Hello, OpenSocial 这一字符串。

代码清单 20.16 简单的插件定义文件

```
<?xml version="1.0" encoding="UTF-8"?>
<Module>
    <ModulePrefs title="my gadget">
    </ModulePrefs>
    <Content type="html">
        <![CDATA[
        <div>Hello, OpenSocial</div>
        ]]>
    </Content>
</Module>
```

代码清单 20.17 是一段使用了 OpenSocial 的 JavaScript API 的代码示例。在这段代码中，仅节选了插件定义文件的 Content 元素部分。

代码清单 20.17 OpenSocial 所提供的 JavaScript API 的使用示例

```
<![CDATA[
<script type="text/javascript">
function init() {
    osapi.people.getViewer({fields: ['displayName', 'birthday']}).
execute(function(result) {
        if (!result.error) {
            document.getElementById('content').innerHTML = result.displayName + "'s
birthday is " + result.birthday;
        }
    });
}
gadget.util.registerOnLoadHandler(init);
</script>
<div>Hello, OpenSocial</div>
<div id="content"></div>
]]>
```

OpenSocial 的 JavaScript API 是一种模块化的 API。在 OpenSocial 中，模块被称为 feature（特性）。可以通过插件定义文件的 <Require> 标签来指定所需使用的 feature（参见代码清单 20.14）。无需特地声明就能使用的 feature 则被称为 core feature。core feature 包含有以下四种 JavaScript API。举例来说，代码清单 20.17 中的 gadgets.util.registerOnLoadHandler 函数就属于这类 API。

- gadgets.io
- gadgets.util
- gadgets.Prefs

- gadgets.json

代码清单 20.17 使用了 osapi feature 的 osapi.people.getViewer 函数。因此，必须在插件定义文件中写上 feature="osapi"[①]。

■ 其他 Web API 的调用

在 OpenSocial 插件中通过客户端 JavaScript 调用 OpenSocial 容器的 Web API 时，不会发生跨源限制问题。插件的 HTML 是通过 OpenSocial 容器接收的。

OpenSocial 插件提供了一种机制，使通过客户端 JavaScript 调用任意的外部 Web API 成为可能。这种行为自然是存在跨源限制的，不过由于 OpenSocial 容器充当了代理中转，因此回避了跨源限制问题。这里所要使用的 API 是 gadgets.io.makeRequest 函数。其第 1 个参数是外部 URL，在收到响应时被调用的回调函数则将被传递给其第 2 个参数。

代码清单 20.18 是一个具体的例子。该例中调用了 Twitter 的搜索 API。由于 response.data 是 JSON 格式的字符串，因此需要通过 gadgets.json.parse 函数对其进行分析，并将其转换为对象。也可以改用在第 2 部分中介绍过的原生 JSON 对象，其 JSON.parse 函数的功能和它是一样的。

gadgets.io.makeRequest 函数还支持 OAuth，因此，也可以调用需要进行权限转让的 Web API。虽然调用是通过客户端 JavaScript 进行的，但其实真正调用了 Web API 的是（中转）服务器，因此，其执行原理与图 19.5 所示的方式相当。在执笔本书时，该函数仅支持 OAuth 1.0，不过今后应该也会对 OAuth 2.0 提供支持[②]。

代码清单 20.18　其他的 Web API 调用示例

```
<![CDATA[
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.6.1/jquery.min.js"></script>
<script type="text/javascript">
function doSearch() {
    gadget.io.makeRequest('http://search.twitter.com/search.json?locale=zh&q=' +
$.(#searchText').val(),
        function(response) {
            var content = $('#content');
            gadgets.json.parse(response.data).results.forEach(function(result) {
                content.append('<li>' + result.text + '</li>');
            });
        });
}
</script>
<div>Hello, OpenSocial</div>
<input type="text" id="searchText"></input>
<div onclick="doSearch()">搜索</div>
<ul id="content"></ul>
]]>
```

专栏

OpenSocial 标准文件的阅读方式及 Activity

OpenSocial 的标准文件被分为了多个文件，要弄明白应该从何处开始阅读并不容易。这里介绍一种简单的方法。

OpenSocial 的标准文件大致可以分为以下三类：定义了插件 XML 结构描述与 JavaScript API 的标准文件、定义了 HTTP API（RESTful API 的）的标准文件，以及定义了 OpenSocial 所允许的数据结构的标准文件。其中最有必要了解的是数据结构。数据结构的标准文件又可以分为 Core Data Spec 与 Social Data Spec 这两种。

① 在旧版 API 中函数前缀为 opensocial。而在新版 API 中则变为了 osapi 前缀。

② OpenSocial 目前已经对 OAuth 2.0 提供了支持。——译者注

前者定义了对象 ID 及用户 ID 等基本的数据结构，而 Social Data Spec 则对 OpenSocial 专有的数据结构进行了定义。也就是说，如果想要尽快了解 OpenSocial 的功能，阅读 Social Data Spec 是一条捷径。

与目前的 OpenSocial v1.1 相比，正在制订中的 v2.0 新版中的 Social Data Spec 有一个显著的变化，将 Activity 改为了 Activity Streams。目前（v1.1）中的 Activity 只是一个附有时间戳的文本，它只包含与用户相关的 Feed 信息。

Activity Streams 对人们的活动信息做了进一步结构化处理。Activity Stream 被定义为 ActivityEntry 的集合。在 ActivityEntry 中，含有 actor（谁）、verb（做了什么）、object（行为的对象）、target（结果，对象的现状）等基本元素。Activity Streams 将会成为一种标准的数据交换格式，以描述人们的活动信息。



第 6 部分

服务器端 JavaScript

回顾历史，JavaScript 都是以客户端 JavaScript 为主，以浏览器为主要运行环境的。随着 JavaScript 的流行，服务器端 JavaScript 也逐渐发展起来。本部分将分别介绍 CommonJS 这一服务器端 JavaScript 的标准 API，以及 Node.js 这种具有代表性的具体实现。

第21章 服务器端 JavaScript 与 Node.js

本章将介绍服务器端 JavaScript 的动向以及 CommonJS 这一标准 API。之后，还会说明服务器端 JavaScript 中发展最为迅猛的 Node.js 的基本信息。Node.js 基于 v8 这一高速的 JavaScript 实现，并且可以以异步处理的方式来描述可扩放的服务器。

21.1 服务器端 JavaScript 的动向

如今，在服务器端使用 JavaScript 的做法正越来越受到人们的关注。回顾过去，我们会发现历史上曾经出现过服务器端 JavaScript，不过之后被废止了。过去，网景公司开发的收费 Web 应用服务器 Netscape Enterprise Server，对通过服务器端 JavaScript 开发 Web 应用提供了支持。随着网景公司的衰弱，服务器端 JavaScript 技术也退出了公众视线。之后，Java 与 .NET 成为了主流的商业服务器端开发语言，而 Perl、PHP、Ruby 及 Python 等则是主流的开源服务器端开发语言。

今天，服务器端 JavaScript 再次受到瞩目。此前，只有网景公司推出了相关产品，而现在已经出现了各种各样的具体实现。

JavaScript 语言的繁荣，自然是服务器端 JavaScript 逐渐繁荣的一个重要原因。正如本书第 1 部分所述，AJAX 与 HTML5 的出现为此提供了基础。而另一个重要原因，则是运行于 Java 虚拟机（JVM）中的 JVM 语言的发展。在 Java6 中，Java 脚本 API 成为了一种标准 API。Rhino 这一移植至 JVM 中的 JavaScript 实现，为服务器端 JavaScript 的运行提供了保证。JVM 语言的优点在于它可以使用 Java 的一些特性。一些无法通过 JavaScript 实现的功能，可以借助 Java 完成。与已经广泛用于服务器端的 Java 相结合之后，在服务器端使用 JavaScript 进行开发的阻碍就减小了很多。

之后出现了 Node.js。它的特点是使用了 v8 这一 Google 公司开发的高速 JavaScript 实现，并支持异步网络处理。通过 Node.js 可以开发出可扩放性很强的 Web 应用，因此它受到了广泛关注。

此外，我们也不能忽略处于云计算中心的 Google 公司所开发的 Google Apps Script 技术。今后，应该会有越来越多的 Web 应用，选择服务器端 JavaScript 作为其扩展语言。

21.2 CommonJS

21.2.1 CommonJS 的定义

正如第 3 部分所讲，在客户端 JavaScript 中，已经有 DOM 这一扩展 API 的事实标准。而在服务器端 JavaScript 中，并没有这样的标准。目前，大部分服务器端 JavaScript 都提供了其自定义的 API 实现。在服务器端 JavaScript 中甚至不存在标准的文件读取 API，这在其他程序设计语言中是必然存在的功能。纷繁的自定义 API 对于开发者和库的使用者来说，都是一种负担。因此，人们开始制订 CommonJS 这一标准 API。不过 CommonJS 只会制订 API 的标准，之后以 CommonJS 为标准的 JavaScript 实现将会根据这一标准提供相应的 API 具体实现。

CommonJS 出现于 2009 年初，是一种新的技术。最早它使用了 ServerJS 这一名称。如名称所示，它的目的是制订出一种服务器 JavaScript 的标准 API。之后，诸如通过 JavaScript 实现的命令行工具或 GUI 工具等服务器端之外的用途也被加入，于是 ServerJS 改名为 CommonJS。大家可以在以下的站点中获取更多信息。

http://www.commonjs.org/

http://wiki.commonjs.org/wiki/CommonJS

CommonJS 还有一种名为 JSGI 的相关标准。CommonJS 的 API 提供了包括文件操作等基本功能，相当于 Java 中 java.lang 及 java.util 包中所涉及的 API。而 JSGI 则是用于 Web 应用的 API，相当于 Java 中的 Servlet API。

专栏

与宿主对象无关的库

jQuery 与 prototype.js 是两种著名的 JavaScript 库。他们都被用于客户端 JavaScript 中。它们没有提供浏览器 UI 相关的功能，并且只能用于全局对象是 window 对象的情况。

随着服务器端 JavaScript 的繁荣，渐渐出现了一些与宿主对象无关的类。例如，underscore.js（http://documentcloud.github.com/underscore/）就是其中的代表之一。

21.2.2　CommonJS 的动向

如今，很多库和框架都是以 CommonJS 为标准的，在服务器端 JavaScript 领域也是如此。本章将要介绍的 Node.js 便是其中之一。此外，JVM 类型的服务器端 JavaScript 中的 Narwhal 及 RingoJS，支持 JavaScript 实现的文档型数据库 CouchDB 等，也都符合 CommonJS 标准。其他的一些客户端 JavaScript 框架，如 SproutCore 也表示将会遵循 CommonJS 标准。

不过，2009 年初，CommonJS 的标准化进程并非一帆风顺，进展顺利的只有接下来将要介绍的模块 API 标准，出现了很多模块 API 的具体实现。但对于其他的 API，无论是标准的标准化，还是具体实现的开发，都没有什么进展。更为糟糕的是，服务器端 JavaScript 中发展势头最好的 Node.js 原本是以异步 API 为核心的，因此，很难和 CommonJS 中传统的同步 API 保持统一。

目前，主流的具体实现（如 Node.js 及 underscore.js 等）中的 API，以及来自于客户端（如 jQuery 及 HTML5 等）的 API，构成了服务器端 JavaScript API 的事实标准，CommonJS 应该也会在今后逐步吸收这些 API。从长远来看，一定会出现 CommonJS 这样的标准。

21.2.3　模块功能

在 CommonJS 中，只有模块功能的标准化及其具体实现的开发工作取得了进展。本节将先说明需要有标准化模块的原因。之后还会介绍 CommonJS 中模块 API 的使用方法。

■ 需要模块功能的原因

如果程序设计的规模达到了一定程度，则必须对其进行模块化。模块化可以有多种形式，但至少应该提供能够将代码分割为多个源文件的机制。

在客户端 JavaScript 中，我们可以通过 HTML 的 script 标签来读取多个 JavaScript 文件。虽然也可以将其理解为一种模块化机制，但这并不是 JavaScript 语言所提供的功能。从程序设计语言的角度来看，通过 script 标签读取文件，其实只是把文件连接了起来而已。此即所谓的包含（include）机制。

通过这一机制将 JavaScript 代码分割为多个文件后，可能会因为全局命名空间受到污染而产生名称冲突的问题。于是，随着具有依赖关系的外部库的增加，这一问题将变得越来越难以处理。对于客户端 JavaScript 来说，我们可以通过在 6.7.4 节中所介绍的方法，以及对每一个库使用不同的名称空间，来解决这个问题。

在服务器端 JavaScript 中，我们也可以通过包含的方式，将多个源文件连接起来。不过，随着文件数量的增加，通过包含进行连接的方式将变得难以管理。为了解决这一问题，我们可以使用 CommonJS 的模块功能。CommonJS 中模块的区分单位取决于具体的实现方式。不过通常都会以文件为单位。本书将说明以一个模块对应一个文件的原则。

■ 模块的具体示例

本书中的说明基于 CommonJS 模块标准 1.1.1 版。

在默认情况下，CommonJS 模块不会对全局命名空间造成污染。在结合两个源文件时，只要其中某一个文件没有输出其变量名及函数名（之后将会详述），这些名称对于另一个文件来说就是不可见的。

本节之后将依次介绍模块部分代码的书写方式及读取模块的代码的书写方式。在作为模块读取的代码中，会预先创建 exports 与 module 这两个对象。由于所有文件都可以作为模块读取，因此其实每个文件中都隐式地创建了 exports 与 module 对象。为了使这些文件能够被作为模块使用，必须设置这些对象的属性。

exports 对象的属性名称是对模块外部公开的。我们可以按照代码清单 21.1 的方式来使用这些属性（假定该文件的文件名为 calc.js），可以在源代码中的任意位置对 exports 的属性进行赋值。

代码清单 21.1　将文件作为模块调用（calc.js）

```
// 设置模块中向外部公开的名称
exports.sum = sum;

function sum(a, b) {
    return Number(a) + Number(b);
}
```

这样一来，我们就能在其他文件中调用 sum 函数了（本节之后将说明使用模块的范例代码）。

在 module 对象中，有两个只读属性 id 与 url。它们的值都与模块名称相同。模块名称被用于识别模块。如果模块以文件为单位，在这一前提下，可以认为模块名就等同于文件名。目前，通过 id 属性的值来获知模块名称是可行的。

■ 在代码中使用模块

本节将说明如何使用模块。我们可以通过 require 函数来使用模块。例如，可以像下面这样来使用前一节中的 calc.js。需要将模块名传递至 require 函数。关于 console.log 的相关说明，请参见 21.3 节。

```
// 在代码中使用模块的示例
var calc = require('calc');
console.log(calc.sum(4, 5));
```

我们也可以以下面这样的方式来使用（不过这并不是模块特有的用法，而是 JavaScript 所具有的用法）。

```
// 在代码中使用模块的示例
var sum = require('calc').sum;
console.log(sum(4, 5));
```

将模块名传递给了 require 函数之后，函数将会查找与之对应的文件。其查找方式取决于具体的实现。是否要在传递给 require 函数的模块名之后添加扩展名 js，也是由具体的实现决定的。在 Node.js 这样的实现中，是否添加扩展名都没有问题，而在有些实现中，则会发生错误。文件的查找路径也取决于

具体实现。有些实现会在当前文件夹中查找，另一些则不是这样。

如果是在当前文件夹中查找文件，只要将 calc.js 文件置于当前文件夹中，就能够通过 require('calc') 对其进行读取。如果没有默认在当前文件夹中查找（Node.js 便是如此），则不能使用 require('calc')，而要改写为 require('./calc')。如果将文件放在了名为 sub 的子文件夹中，则需要写成 require('sub/calc') 或 require('./sub/calc')。

如果没有找到所指定的模块，require 函数将会抛出一个 Error 异常对象。一个模块可以 ruquire 另一个模块。即使模块间进行了循环调用，也不会有任何问题。

■ **模块的应用**

require 函数（Function 对象）有两个属性，分别是 main 与 paths。可以像下面这样，将 main 属性与 module 对象作等值比较，以判断该文件是否是被直接执行，或者以模块的形式被读取。

```
if (require.main === module) {      // 如果文件是被直接执行的，则为真
    console.log('directly called');
} else {                            // 如果文件是被作为模块来读取的
    console.log('loaded as a module');
}
```

require.paths 属性的值是一个数组，它含有模块的查找路径。可以通过下面的手段，在运行时更改模块文件的查找路径。不过，我们并不推荐这种做法（至少在 Node.js 中不推荐）。不过，在当前代码中仍然有这一功能，因此我还是会介绍一下。

```
// 不推荐使用这种方法
// 在运行时，更改模块文件的查找路径
require.paths.unshift(_dirname + '/subdir');     // _dirname 是 Node.js 所特有的
```

21.3 Node.js

21.3.1 Node.js 概要

本节将介绍 Node.js 这种服务器端 JavaScript。本书第 4 部分的 WebSocket 对此也略有提及。Node.js 是一种以异步处理为特点的 JavaScript 实现 [①]。其站点的 URL 如下。

http://nodejs.org

Node.js 具有以下特征。

- **以 v8 作为 JavaScript 实现（至于网络方面，则使用了 libev 或 libeio 等已有的 C 库）。**
- **在实现中提供了通用的异步处理事件循环。**
- **附带支持会话式壳层的命令行工具。**
- **可以通过包系统扩展。**

最近，在提到服务器端技术时，我们指的常常是 Web 应用层。相较于通常的 Web 应用服务器层，Node.js 是一种处于更低层的软件。

其他的服务器端 JavaScript，通常都能够在 Apache 等已有的 Web 服务器，或 Tomcat 等已有的 Java Servlet 引擎中运行。然而，在通过 Node.js 开发 Web 应用时，Node.js 可以覆盖从 HTTP 服务器层至 Web 应用服务器层的范围。下一章还将介绍 Express，通过这一方式，我们甚至能够使用 Node.js 来构建 Web 应用

① 根据 FAQ 的说明（https://github.com/joyent/node/wiki/FAQ），其正式名称既不是 node，也不是 Node.js，而是 Node。不过，为了提高可识别性，也能够使用 Node.js 这一名称。在本书中，将统一使用 Node.js 的称法。

程序框架。因此，理应将 Node.js 定位为一种与 Perl 或 Ruby 等脚本语言处于同一层面的软件。将库层置于其核心的 JavaScript 实现 v8 之上，并结合 node 这一命令行工具，就能使 Node.js 获得等同于 Perl 或 Ruby 这样的脚本语言的功能（当然，要达到成熟还有很大差距）。

本书将以 Node.js Web 应用开发为中心，介绍服务器端 JavaScript。无论是邮件服务器还是名称服务器，是命令行工具还是 GUI 工具（假如绑定了 GUI 库），都可以通过 Node.js 实现。

本书的内容是基于 Node.js 版本 0.4.8 写成的。对 Node.js 安装方法的说明就此省略。在 tar.gz 类型的源文件中同时包含了所需的库（v8 及 libev 等）的源文件。在 GNU/Linux 中，我们可以通过 configure 与 make 来编译。

■ 第三方模块

围绕着 Node.js，有很多活跃的开发社区。关于具有代表性的第三方模块，大家可以参见下面的 URL。

http://github.com/ry/node/wiki/modules

很多第三方模块都是以包为单位发布的。关于包的详细信息，本章之后将会和 npm 一起说明。

■ Node.js 的 API

为了便于理解，本章在介绍 API 时，将使用“类”这一术语。相关术语的用法，请参见 2.5.7 节。

在对类进行说明的参考文档中，通常都会记载类属性（域以及方法）的一览表。在 Node.js 的文档中，还包括了类与对象所能触发的事件，以及事件所对应的回调函数的参数信息等说明。目前，除了阅读源代码，只能通过参考文档来了解这些信息。

我们可以通过下面的 URL，获得 Node.js 的 API 参考文档。

http://nodejs.org/docs/latest/api/index.html

该 API 文档仅介绍了宿主对象及标准发布版中所含的模块 API。得益于丰富的外部模块，Node.js 有着强大的扩展性。关于这些外部模块的信息，则需要参见其各自的 API 参考文档。

■ 异步处理与非阻塞处理

在此，我们来整理一下 Node.js 中出现的一些容易混淆的术语。

首先是阻塞与非阻塞这两个术语。这两个术语可以用于描述处理的调用方式，通常被用于函数调用方式的描述中。

阻塞式函数可以在函数内保持等待状态。

与之相对地，非阻塞式函数没有等待状态。在获取某些资源时，将会进入等待状态。从运行层面来看，I/O 等待于锁（lock）等待是常见的等待类型。

同步处理与异步处理的含义与其所描述的对象相关。对于 Node.js 来说，可以将其理解为用于描述 I/O 处理的术语。同步 I/O 处理在开始读取或写入操作后，就无法进行其他操作，直至处理结束。在进行同步 I/O 处理时，读取操作常常会引起等待。话虽如此，在进行写入操作时，通常会将数据写入缓冲区内存。这种方式先在表面上完成了处理，从而使性能获得了提升[①]。而在缓冲区写满后是否需要等待，则取决于该函数是阻塞式函数还是非阻塞式函数。

异步处理则会在后台进行读写操作[②]。狭义的异步读取操作，会在读取完成时触发事件。这时，所指定的数据已经被读取至了内存。而广义的异步读取操作，将在能够读取时触发事件，应用会在处理该事

---

① 当然，其内部还需要一定的时间真正完成处理。——译者注

② 对于应用程序来说，这里的“后台”一词，有时指的是操作系统（内核），有时指的是程序中其他的线程。在 Node.js 中，执行异步 I/O 处理的后台是操作系统（内核）。

件的过程中进行读取操作。

异步写入操作将会准备一块内存空间，作为写入数据的缓冲区。在写入完成时，将会触发事件。这一事件也隐含了缓冲区内存中存在空闲空间，可以写入新的数据的含义。广义的异步写入操作，将会在可以进行实际的写入操作时触发事件，并在事件的处理过程中进行写入操作。

下面，我们来总结一下以上关系。网络 I/O 处理本质上是一种具有等待状态的处理。因此，通过同步 I/O 执行网络处理的函数，其实是一种阻塞式的函数。同步的网络 I/O 处理就等同于阻塞式函数处理，这是必然的情况。

另一方面，异步 I/O 处理的含义则更为广泛，实现方式也较为多样。通常，异步数据读写函数都是非阻塞式的。不过，理论上，将数据读写状态的检查函数（peek 操作）与阻塞式的读写函数相结合，也能够实现异步 I/O 处理。因此，在异步 I/O 处理中并非必须要用非阻塞式的函数。正如之前所说，事件被触发时的实现方式也有多种类型。不过，在实际中，我们通常都会将异步 I/O 处理与非阻塞式函数作为同一种概念使用。

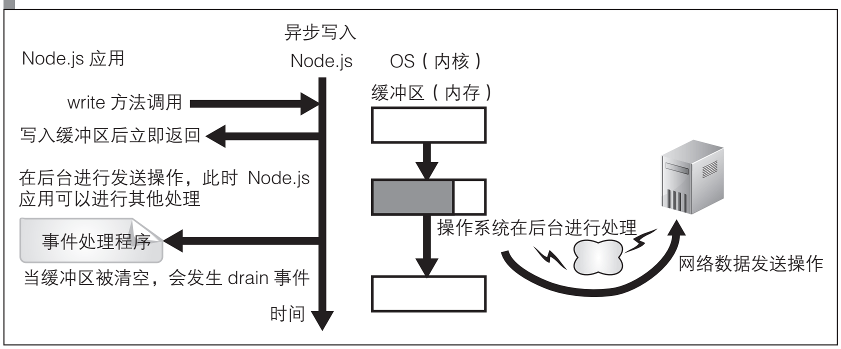
对于网络处理这类必然存在等待的情况，如果要实现并行处理，则必须使用多线程或异步处理等机制。

■ Node.js 中异步处理的执行方式

先不考虑概念上的问题，仅从其实现方式来看，异步处理中的读取操作和写入操作是不对称的。这是因为，在异步处理中通常会使用一块内存空间来作为读写缓冲区。此时，将以内存在读写操作中不会发生阻塞为前提进行讨论。

在 Node.js 中，异步写入处理是通过方法调用来执行的（这些方法通常名为 write）。方法在被调用后，基本上不会发生阻塞就能立即返回。也就是说，这些是非阻塞式的方法（函数）。实际的写入操作是在后台异步进行的。对于网络来说，则会进行数据发送操作（图 21.1）。

图 21.1　异步 I/O 处理（写入）

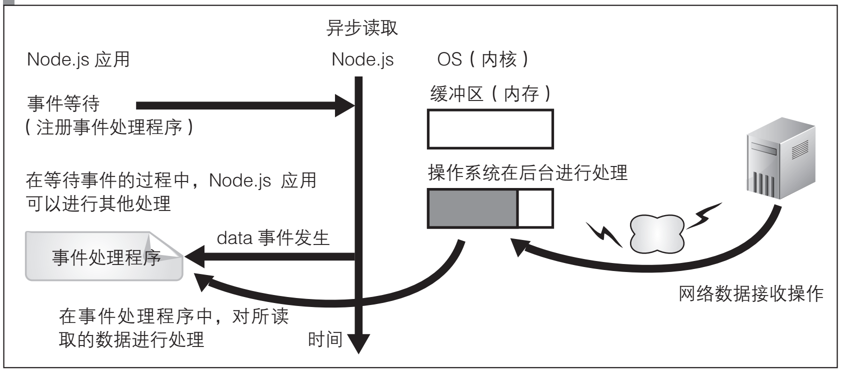


认为只要在写入操作结束时触发事件，并在事件中进行下一个写入操作即可的想法，是一种很常见的错误。这种方式的效率十分低下。正确的做法是，在写入操作完成前，就直接依次调用写入方法（函数）。不必在意实际在何时执行写入操作的，只需将数据写入缓冲区即可。不过，如果所写入的数据非常大，情况则会有所不同。如果写入操作始终比实际的发送操作更快，缓冲区体积就会越来越大，可能会占据过多的内存空间。这种情况下，就需要进行一些复杂的控制处理，要暂停对写入处理的调用，并等待缓存清空的事件发生。

在 Node.js 中，没有（诸如 read 之类的）与异步读取处理相对应的函数或方法。非要说的话，只有 resume 这类的方法，它的作用是用于解除读取事件的等待。读取操作是通过事件所对应的回调函数来执行的。该事件名通常为 data。在后台完成了读取操作之后，数据就进入了内存，这时将会调用回调函数。回调函数的参数将接收所读取的数据。如果读取操作还在进行，则会继续依次调用回调函数。这时会发

生一种奇怪的反转现象。通常来说，在函数调用时，传递给函数的参数是输入值，而函数的返回值是输出值。不过，在通过回调函数进行异步处理时，回调函数的参数才是该处理的输出值（图 21.2）。

图 21.2 异步 I/O 处理（读取）



以上就是 Node.js 中异步处理的原理，这些操作都将会遵循这里所说规则，请大家好好掌握。本章的后半部分将会介绍这些操作的具体示例。

■ 模块

Node.js 中模块的基本机制遵循了 CommonJS 的模块标准，其模块的基本使用方式（exports 及 require）也是符合该标准的。详细内容请参见前一节。不过在 Node.js 中，文件的读取路径等方面则是由具体实现决定的，本节将对其进行说明。

如表 21.1 所示，我们可以用三种方式在 require 的参数中书写并指定模块名称。

表 21.1 Node.js 中模块名的指定方式

名称	说明	示例
相对路径	以 ./ 或 ../ 起始的路径	./foo 或 ./foo.js
绝对路径	以 / 起始的路径	/foo 或 /foo.js
名称	以上两种情况以外的方式	foo、foo.js、foo/bar 或 foo/bar.js

模块名与文件名的对应关系将会以下面的顺序匹配，并载入首先找到的文件。是否在模块名之后书写扩展名不会影响操作。

- 完全一致的文件名
- 添加了扩展名 js 的文件名
- 添加了扩展名 node 的文件名（将会以二进制模块的形式载入）

被指定了相对路径的 require 函数，将会从源文件所在的目录开始，以相对路径搜索文件。如果指定的是绝对路径，则会以模块名寻找与该绝对路径相匹配的文件。如果既没有使用相对路径，也没有使用绝对路径，则会以下面的顺序查找文件。

- 如果与核心模块的名称相一致，则载入核心模块
- 在当前目录下的 node_modules 子目录中查找
- 在当前目录上级的 node_modules 目录中查找

（例如，当前目录为 /home/inoue/nodejs/work/，则将以下面的顺序查找

/home/inoue/nodejs/work/node_modules

/home/inoue/nodejs/node_modules

/home/inoue/node_modules

/home/node_modules
/node_modules)

- 环境变量 NODE_PATH 所指定的目录
- $HOME/.node_modules/ 目录
- $HOME/.node_libraries/ 目录
- Node.js 的安装目录，即 /lib/node/ 目录

模块中所写的代码将会在调用 require 时执行。不过，即使多次调用了 require，也只会执行一次。require 所返回的也始终是同一个对象引用。

■ module.exports

除了代码清单 21.1 中的写法，在 Node.js 中还常会以下面的方式来使用模块。

```
// 和代码清单 21.1 的功能相同

module.exports = {
    sum: function(a, b) {
        return Number(a) + Number(b);
        }
    // 如有需要，可以继续添加其他属性
};
```

21.3.2　node 指令

node 指令是用户所能实际接触到的 Node.js 实体。不以任何参数执行 node 指令，就能够启动一个会话式的壳层。在启动了这一会话式壳层之后，就进入了下面这样的待输入界面。

```
$ node
>
```

可以在这里输入 JavaScript 语句，在按下 Enter 键后将会返回语句的执行结果。如果输入的是表达式语句，则会显示表达式的求值结果。

```
$ node
> 1 + 2
3
> console.log('hello');        // console.log 的含义将在下一节说明
hello
```

在末尾不添加分号也能正常执行，不过本章的示例将不会省略分号。之后，将会以同样的形式来表示 node 的会话式壳层的执行结果。

■ 文件的执行

如果将文件名传递给 node 指令的命令行参数，就能够执行文件内的 JavaScript 代码。其使用方式与 Perl 或 Ruby 等所谓的脚本语言相似。

```
$ node my.js
```

在上面的例子中，文件能被执行的前提是，写有 JavaScript 代码的 my.js 文件位于当前目录下。只要在 my.js 之前写上 #!/usr/bin/node 这样的 node 指令路径，就能设置执行权限位，从而可以直接运行文件。

21.3.3　npm 与包

在 Node.js 中，有一种被称为 Node Package Manager（npm）的包系统。npm 的官方站点 URL 如下。

http://npmjs.org

可能有人会对模块与包之间的关系感到迷惑，本节将解释两者的功能差异。模块是由 Node.js 提供的功能，借助这一语言功能，我们能够将程序分为若干部分。具体来说，它可以将源文件分割为多个文件以便管理。进一步来说，这是一种能够分割命名空间的语言功能。

另一方面，包是一种软件发布机制。通常，包被定义为模块的集合。理论上来说，也可以不通过模块的形式来构建包。而会对 Node.js 宿主对象的命名空间产生影响的包也是能够被创建的。不过，这种包的性质并不良好，因此，不应该创建这种类型的包。

我们可以向下面这样安装 npm[①]。

```
$ curl http://npmjs.org/install.sh | sh
```

安装完成后，就能够使用 npm 指令了。npm 指令的基本功能是搜索包、安装及卸载操作。关于 npm 指令的使用方法，请参见通过 npm help 输出的在线帮助。还可以通过下面的 URL，搜索已有的 npm 包。

http://search.npmjs.org/

我们可以像下面这样，通过 npm 指令来安装一个包。

$ npm install 包名

按照上面的方式安装的话，将会以当前目录为基础，在相对路径中安装所指定的包。像下面这样添加 -g 选项之后，就会执行全局安装。

```
$ npm install -g 包名
```

本章将在必要时进一步说明通过 npm 来安装包的方法。

21.3.4 console 模块

console 对象的内部是一个 console 模块的对象。默认情况下，该模块将会被载入，因此，不需要显式地使用 var console = require('console') 语句来执行这一载入操作[②]。

表 21.2 列出了 console 模块中的函数（从形式上来说，这些是 console 对象的方法）。这些函数主要被用于调试之中。它们的实现并不复杂，只要查看 Node.js 源文件树中的 console.js 文件就能理解。

表 21.2 console 模块的函数

函数名	说明
log(a,b,⋯)	以标准规格输出所有参数，并在所输出的消息末尾再输出一个换行符
info(a,b,⋯)	目前只是 log 函数的一个别名（将来可能会具有不同的功能）[③]
error(a,b,⋯)	以标准错误规格输出所有参数，并在所输出的消息末尾再输出一个换行符
warn (a,b,⋯)	目前只是 error 函数的别名
assert(expr)	如果参数 expr 的值为假，则抛出 AssertionError 异常
dir(obj)	检查对象
trace(label)	以标准错误规格输出调用栈。label 将用于输出消息之中
time(label)	开始性能分析（测量执行时间）
timeEnd(label)	结束性能分析。其输出结果为，从具有相同标签值的 time 函数被调用起至今所经过的时间。会通过 log 函数输出这一结果

对于消息输出类的函数，可以使用表 21.3 中的格式指定符。只要在函数第 1 个参数的字符串中写上格式指定符，就能够以一定的格式，将其与之后的参数进行替换。

① 在此之前，请先完成 Node.js 的安装。

② 从语句含义的角度来看，这行语句的功能相当于载入一个 console 对象。

③ 事实上，info 是 log 的一个子类别，表示的是“一般信息”。——译者注

表 21.3　消息输出函数的格式指定符

格式指定符	说明
%s	将参数转换为字符串型并输出
%d	将参数转换为数值型并输出
%j	将参数转换为 JSON 并输出（其内部使用了 JSON.stringify[①]）

图 21.3 是含有格式指定符的 console.log 的使用示例。

图 21.3　console.log 的使用示例

```
> console.log('foo');
foo
> console.log('foo', 'bar');
foo bar

> var n = 7;
> console.log('%d foo', n);
7 foo
> console.log('%d foo', n, 'bar');
7 for bar

> var obj = {x:1, y:2};
> console.log(obj);        // 如果传递的是对象引用，则会以 JSON 格式输出
{ x: 1, y: 2 }
```

console.dir 函数将会显示所指定对象的属性一览，这就是所谓的检查函数。从内部来看，它输出的是 util.inspect 函数的结果。ECMAScript 第 5 版存在 Object.keys 或 Object.getOwnPropertyNames 这样的标准检查 API（参见表 5.7）。因此，并非一定要使用 console.dir 才能检查对象的属性。不过，console.dir 的输出格式非常整齐明了，请根据情况选取使用这两种方式。

21.3.5　util 模块

util 模块和 console 模块一样，都是能使调试过程更为简便的模块[②]。如果要使用 util 模块，则需要像下面这样显式地载入该模块。可以以任意的变量名来指称这一模块，不过，如无特别理由，最好使用 util 这一名称，这样的可读性较高。

```
// util 模块的使用方法
var util = require('util');
util.print('foo');            // 调用 util 模块内的函数
```

表 21.4 列举了 util 模块内具有代表性的一些函数。

表 21.4　util 模块内具有代表性的函数

函数名	说明
print(a,b,…)	将所有参数转换为字符串型，并以标准输出规格显示
puts(a,b,…)	将所有参数以标准输出规格显示
debug(msg)	将参数以标准出错规格显示
error(a,b,…)	将所有参数以标准出错规格显示
inspect(obj, showHidden, depth, colors)	返回一个字符串，该字符串将会以一定的格式显示对象的属性一览。其他参数的说明在此略去
log(msg)	以标准输出规格显示当前时刻及相关参数
inherits(ctor, superCtor)	第 1 个参数中的类对象，将会原型继承第 2 个参数的类对象

util.inherits 的使用方式如代码清单 21.1 所示。其内部的执行方式与本书第 2 部分 5.17.2 节中所例举的代码是相同的。通过 util.inherits，我们可以方便地为自定义的对象增加事件分发功能。21.3.9 节将对详

① 请参见本书第 2 部分。

② 如今 util 模块已经取代了 sys 模块的地位。不过为了向下兼容，sys 模块仍然存在。在实际使用中还请选用 util 模块。

细内容进行说明。

代码清单 21.2 util.inherits 的使用示例

```
// 相当于基类的构造函数
function Base() {
Base.prototype.method = function() { console.log('base method called'); }

// 相当于派生类的构造函数
function Derived() {}

// 继承
require('util').inherits(Derived, Base)
```

```
// 代码清单 21.2 的使用示例
> var obj = new Derived();
> obj.method();
base method called
```

21.3.6 process 对象

process 对象是一种宿主对象。和 console 对象一样，它仅有一个实例，即所谓的单例对象（Singleton）。全局变量 process 是默认存在的，可以通过它引用 process 对象。可以通过 console.dir(process) 来显示 process 对象的属性一览。表 21.5 列出了 process 对象中一些具有代表性的属性。

表 21.5 process 对象的属性（节选）

属性名	说明
argv	命令行参数。字符串值数组。数组的第 1 个元素为 'node',第 2 个元素为执行文件的路径。从第 3 个参数起是命令行选项
env	以关联数组形式保存环境变量的对象
stdin	标准输入格式的数据流对象
stdout	标准输出格式的数据流对象
stderr	标准出错格式的数据流对象
exit(status)	结束进程。参数 status 是进程的错误代码
version	诸如 'v0.4.8' 这样的字符串值
platform	诸如 'linux' 或 'win32' 这样的字符串值
pid	进程 ID 的数值
cwd()	以字符串形式返回当前目录
chdir(dir)	将当前目录更改为指定目录
getuid()	返回当前运行用户的用户 ID
setuid(uid)	用指定的用户 ID 更改当前运行用户
kill(pid, signal)	将信号发送给指定 pid（数值）的进程。信号是以类似于 'SIGTERM' 形式的字符串值指定的。

关于 stdout、stdin 及 stderr，之后的 21.3.11 节将对此进行说明。process 对象将会触发表 21.6 中的这些事件。

表 21.6 process 对象的事件

事件名	事件处理程序	说明
exit	function(){}	该事件将在进程结束时被触发
uncaughtException	function(exception){}	该事件将在抛出了没有被捕获的异常时被触发。事件处理程序的参数是一个异常对象
信号字符串	function(){}	诸如 'SIGINT' 这样的字符串

例如，可以像代码清单 21.3 这样来设计 uncaughtException 事件的事件处理程序。其中 on 的含义将会在之后说明。

代码清单 21.3　uncaughtException 事件的使用示例

```
process.on('uncaughtException', function(err) {
    console.log('Got an error: %s', err.message);
    process.exit(1);
});

throw new Error('foo');
```

可以在 process 对象的事件处理程序中使用操作系统级别的信号处理程序。例如，下面的代码将会响应 SIGINT 信号。

```
process.on('SIGINT', function() {
    console.log('Got SIGINT signal');
});
```

21.3.7　全局对象

如果不太清楚全局对象的含义，可以重新阅读一下本书 5.21 节。在 Node.js 中，全局变量 global 是默认存在的。该 global 变量的作用与客户端 JavaScript 中的 window 变量相同，始终指向了全局对象。

可以像下面这样，获得所谓的全局变量与全局函数一览表。

```
> Object.keys(global);     // 在最外层代码中也可以使用 Object.keys(this)，两者的功能相同
[ 'global',
    'process',
    'GLOBAL',
    'root',
    'Buffer',
    'setTimeout',
    'setInterval',
    'clearTimeout',
    'clearInterval',
    'console',
    'module',
    'require',
    '_' ]
```

在此略去输出结果。而通过 Object.getOwnPropertyNames(global)，则能够获得更全面的全局信号一览。

21.3.8　Node.js 程序设计概要

■ 回调函数

Node.js 程序设计中的核心就是异步处理。由于 Node.js 的代码是以异步处理的方式执行的，因此经常会使用回调函数。例如，下面代码的能够在 3 秒之后以标准输出规格输出一段消息。

```
// 3 秒后输出至控制台
setTimeout(function() { console.log('time up'); }, 3000); // 通过参数来指定回调函数
```

上面的代码也可以在会话式的壳层中执行。不过，在通过壳层执行时，其执行方式可能较难理解，所以请先准备好写有以上代码的 time.js 文件，然后像下面这样通过 node 指令执行。执行结果如下所示。

```
// 通过命令行执行以下指令
$ node time.js
// 3 秒之后将会输出以下内容，同时 node 指令全部完成
time up
$
```

■ 事件循环

异步处理的价值能够在网络处理中得到最大的发挥。接下来，我将介绍一段简单的通过 Node.js 来实

现的 HTTP 服务器处理的代码（代码清单 21.4）。这段代码将会通过 8080 端口接收 HTTP 请求，并将 HTTP 响应返回至发送者。在 Node.js 中，只要书写了事件等待的代码，就会隐式地在脚本中加入事件循环，之前 setTimeout 函数的例子也是如此。如果没有事件等待代码，脚本就会在全部运行完成之后结束。

代码清单 21.4 中的代码，首先通过 listen(8080) 创建了一个等待连接的事件源（即一个能够触发事件的对象），然后在代码的末尾隐式地加入了一个事件循环（需要注意的是，该循环不会在 listen 被调用时中止）。如果在 8080 端口收到了 HTTP 连接，就会在事件循环中调用回调函数。回调函数将会处理 HTTP 响应。在离开了回调函数之后，它将会再次回到隐式的事件循环中。

代码清单 21.4　通过 Node.js 实现简单的 Web 服务器

```
var http = require('http');

http.createServer(function(request, response) {      // 通过参数来指定回调函数
    response.writeHead(200, {'Content-Type': 'text/plain'});
    response.write('Hello ');
    response.end('World\n');
}).listen(8080);
```

代码清单 21.4 中有很多隐式的处理，本节之后将介绍其对应的显示代码（代码清单 21.5）。可以通过 on 函数在 httpd 对象中设定回调函数。on 函数是 Node.js 程序设计中最为重要的函数，需要通过该函数来对事件设定回调函数。这类回调函数也被称为事件处理程序。从内部来看，on 函数是 addListener 函数的一个别名（之后将会详细说明）。

代码清单 21.5 中的 on 函数，对 httpd 对象的 'request' 事件设定了事件处理程序。httpd 对象的 'request' 事件将会在收到 HTTP 请求时被触发。在 Node.js 中，我们通常会通过事件的名称来识别事件，并将事件与事件处理绑定。之前代码清单 21.4 中的回调函数，从内部来看，也是 'request' 事件的事件处理程序，只不过事件名被隐藏了而已。

代码清单 21.5　通过 Node.js 实现简单的 Web 服务器（显示地使用回调函数）

```
var http = require('http');

var httpd = http.createServer();
httpd.listen(8080);

httpd.on('request', function(request, response) {        // 通过参数来指定回调函数
    response.writeHead(200, {'Content-Type': 'text/plain'});
    response.write('Hello ');
    response.end('World\n');
//  httpd.close();    // 如果去除了注释标记，Web 服务器就会在返回了 HTTP 响应之后终止
});
```

■ 事件处理的格式

如同代码清单 21.5，向事件添加事件处理程序的方式，是 Node.js 程序设计中的基本内容。这即是所谓的事件驱动程序设计风格。关于事件驱动的详细内容，可以参见 6.8 节。

仅从格式上来理解 on 函数是很容易的。只要以下面的方式理解即可。

```
// on 函数的格式
发生事件的对象 .on(' 事件名 ', 事件处理程序 );
```

要理解这一格式并不难，不过，很可能也会产生以下疑问。

- 发生事件的对象是怎样的对象
- 能够使用那些名称来作为事件名
- 事件处理程序会在何时、被谁、以什么参数调用

这些问题将在之后的小节中得到解答。

21.3.9　事件 API

首先，我们来整理一下术语。在响应事件时所调用的回调函数被称为事件处理程序或事件侦听器。在 Node.js 中，这两个术语经常会被混用，本书将统一使用事件处理程序一词。

事件的发生，也称为事件的发出（emit）或触发（fire）。在本书中，如果事件一词作为主语，则统一使用发生这一术语，如果事件一词作为宾语，则统一使用发出一词[①]。大多数事件都有其发出者对象。也有一些诸如计时器事件这样的，发出者不明确的事件。如果将这类事件的发出者看作全局对象，那么所有的事件就都有发出者对象了。发出事件的对象被称为事件源（对象）。

所有的事件都可以通过事件源，以唯一的字符串（事件名）来进行识别。各个事件的名称是由其各自的事件源决定的。如果要了解可以使用哪些事件，则需要阅读 API 参考文档，或阅读 Node.js 的源代码。

事件与事件处理程序之间是 1 对多的对应关系。如果对已经设定了事件处理程序的事件使用 on 函数，原有的事件处理程序并不会被替换，而是会与新的事件处理程序相连。在当前实现中，对于同时存在多个事件处理程序的情况，将会以 on 函数的设定顺序来调用事件处理程序。不过，这样一来，调用顺序就会和代码的具体写法相关联，而这是有违事件驱动程序设计的习惯，因此，应尽可能避免这种做法[②]。此外，目前还无法在中途中断多个事件处理程序的调用链。

■ EventEmitter 类

Node.js 中的事件源继承于 events 模块中的 EventEmitter 类，且其事件继承了表 21.7 中的方法。因此可以认为，在本章的说明中出现的带有 on 的类及对象，都继承于 EventEmitter 类。例如，在代码清单 21.5 中，http.createServer() 返回了一个对象，且该对象实例所对应的类是由 EventEmitter 类继承而来的。

表 21.7　在事件中所使用的方法

函数名	说明
addListener(type, listener)	向名为 type 的事件添加事件处理程序
on(type, listener)	addListener 的别名
once(type, listener)	与 addListener 意义相同，不过此时事件处理程序仅会被调用一次
removeListener(type, listener)	删除所指定的事件处理程序
removeAllListeners(type)	将名为 type 的事件的事件处理程序全部删除
emit(type[, arg,…]	发出名为 type 的事件
listeners(type)	返回名为 type 的事件中所有的事件处理程序
setMaxListeners(n)	设定一个事件所能设定的事件处理程序的上限（默认值为 10）

在表 21.7 中，我们经常会用到的是事件处理程序的添加与删除方法。addListener 与 on 只有名称上的差异，使用哪一个都没有问题。其中 on 较为简短好用，因此，本书将主要使用 on。由于事件的添加与删除方法的返回值是事件源的引用，因此我们可以通过方法链，以 event_source.on('foo', handler1).on('bar', handler2) 的形式调用。

如果在删除方法中使用了 bind，则必须对与引用有关的一些问题加以注意（参见之后对事件处理程序内 this 引用的说明）。而 emit 方法则会在之后介绍自定义事件时说明。

listeners 方法返回的并不是事件处理程序的副本，而是这些事件处理程序（即函数的数组）。因此，如果对 listeners 方法所返回的数组进行添加或删除操作，事件处理程序就会被更改。由于这通常是有违

① 原书中虽然对该术语进行了分类与统一处理，但由于翻译及中文语言习惯的问题，在译文中并没有严格遵循原文。事实上这两种术语的译法都是得获得广泛认可的，无需过分在意对其进行区分。——译者注

② 不应使事件处理程序的调用顺序与代码内容相关，看似是一条很容易遵守的规则，事实上并没那么简单。个人观点是，在遵守这一原则而使代码变得难以理解时，不用过分拘泥于代码的易读性，保留这样的复杂代码即可。不过必须认识到，如果事件处理程序之间存在依赖关系，代码将会变得难以更改。

程序设计规范的，因此尽可能不要采用这种方法。

对于 1 个事件来说，它能够注册的事件处理程序的数量是有上限的。这一机制能够在因错误而不断添加事件处理程序时发挥作用，以便找出问题所在。默认情况下，事件处理程序的数量超过 10 时，就会以 console.error 发出警告。通过 setMaxListeners 方法则能提高该上限值。

EventEmitter 类（及其子类）的对象实例将会发出 newListener 事件。当对象中添加了新的事件处理程序，该事件就会发生。newListener 事件的回调函数的格式如下所示。其中第 1 个参数是事件名，所添加的事件处理程序的 Function 对象则会被传递至第 2 个参数。

```
function(type, listener) { }
```

■ 事件处理程序内的 this 引用

事件处理程序内的 this 引用指向的是事件源对象。其功能与客户端 JavaScript 中 DOM 事件的情况是一样的。

可以通过 bind 来更改事件处理程序内的 this 引用。关于 bind 的详细内容，请参见本书第 2 部分表 6.4 的说明。而如何通过 bind 更改事件处理程序内的 this 引用，6.8 节已经进行过说明。

如果通过 bind 传递了事件处理程序，则必须在使用 removeListener 删除事件时多加小心。下面的代码无法按预期的方式删除事件处理程序。其原因在于，二次调用 bind 时所返回的不是同一个 Function 对象。

```
event_source.on('foo', callback.bind());
event_source.removeListener('foo', callback.bind());// 与上一个回调函数是两个不同的函数
```

■ 在事件处理程序内调用异步处理时的注意事项

本节将列举 Node.js 程序设计中的一些多发问题。

事件处理程序中的代码，将会根据事件的不同而执行不同的处理，这从概念上来说并不难理解。毕竟，如果之前有 GUI 中事件驱动程序设计经验，对这种风格应该会很熟悉，而 JavaScript 程序员大多也会对客户端 JavaScript 中的 DOM 事件模型相当了解。在 GUI 以外的环境中，在程序设计中使用观察者模式或回调函数也是非常常见的。在事件驱动程序设计中需要注意的是，不能在事件处理程序中书写可能造成阻塞的处理（在任何的事件驱动程序设计中都是如此），同时还要对事件处理程序中的 this 引用加以注意（在 JavaScript 中必须小心处理这一问题）。

而 Node.js 中的程序设计模型与传统的事件驱动模型有少许差异。通常，在事件处理程序的代码中，是不能调用异步处理的。在 Node.js 以外的很多事件驱动程序设计中，事件处理程序内所调用的函数与方法也大多是同步的。

而另一方面，在 Node.js 中，我们可以正常地在事件处理程序中调用异步函数及方法。回调函数常常会直接以匿名函数（作为闭包）的形式传递，因此很容易造成误解。接下来，本节将通过具体的例子，列举容易出现的误解。

代码清单 21.6 是 Node.js 的典型错误。其中个别 API 的含义将会在之后的小节逐一说明，其功能是通过 URL 来指定文件，并将文件内容作为响应返回，从而实现了 HTTP 服务器处理。错误处理之类的内容已被省略。代码清单 21.6 的关键之处在于，在事件处理程序中调用了不同的异步处理。外部事件处理程序被注释为了“事件处理程序内 1”，内部事件处理程序则被注释为了“事件处理程序内 2”，请对此加以确认。

在代码清单 21.6 中我们需要注意的是，事件处理程序 2 中读取的文件的内容，被赋值给了事件处理程序 1 中的本地变量 content。readFile 函数的第 3 个参数是一个回调函数，它被作为事件处理程序调用。回调函数是一种闭包，因此（从闭包的角度来看），它可以访问外部的局部变量 content。如果对这部分的执行方式不太清楚，可以参见本书第 2 部分的相关章节。由于文件读取能够被立即完成，因此似乎文件的内容是能够被传递给事件处理程序的参数 str 的。不过，代码清单 21.6 的 HTTP 服务器实际上总会返回 'not excepted' 的响应。

代码清单 21.6　这段代码无法得到预期的执行结果

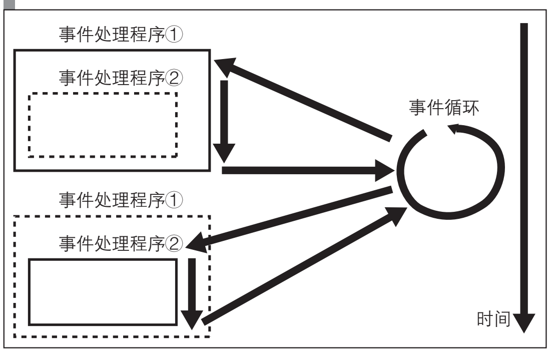
```
var http = require('http').createServer(function(request, response) {
    // 事件处理程序内 1
    var content = 'not expected';

    var filepath = './' + require('url').parse(request.url).pathname;
    // 这条语句有效的前提是所指定的文件存在于当前目录中
    require('fs').readFile(filepath, 'utf8', function(err, str) {
        // 事件处理程序内 2
        if (err) throw err;
        content = str;
    });

    response.writeHead(200, {'Content-Type': 'text/plain'});
    response.end(content);
}).listen(8080);
```

代码清单 21.6 中的代码之所以无法获得预期结果，是因为它事实上将会以图 21.4 所示的方式执行。只要还没有离开事件处理程序①，就无法调用事件处理程序②。因此，事件处理程序②中的 content = str 一行始终会在 response.end(content) 被调用之后才被执行。如果对执行顺序没有概念，可以在代码清单 21.6 中插入 console.log 语句，以确认执行顺序。

图 21.4　代码清单 21.6 的执行方式的图示



■ 事件驱动程序设计中的铁则

为了不发生上一节中所说的错误，我们必须清楚地认识到，所有的事件处理程序都只能在（隐式的）事件循环中被调用。并且，只要尚未离开事件处理程序，就无法回到事件循环中。在开发过程中切勿忘记这一原则。由这一原则还引出了另一个重要的注意事项，即在事件处理程序中，绝对不能发生阻塞。之前说过，这一点是事件驱动程序设计中的一条铁则。在事件处理程序中停止的话，是无法再回到事件循环中的，于是这个程序就将会中止执行。

将代码中的 readFile 替换为同步的 API，确实也可以使代码清单 21.6 能以预期的方式执行。不过，这种解决方法并不符合 Node.js 习惯。正确的做法是，像代码清单 21.7 那样，改写内部的事件处理程序②，在其中写上 HTTP 响应处理的代码。

代码清单 21.7　改写代码清单 21.6 以使其能正常运行

```
var httpd = require('http').createServer(function(request, response) {
    var filepath = './' + require('url').parse(request.url).pathname;
    require('fs').readFile(filepath, 'utf8', function(err, str) {
        if (err) throw err;

        response.writeHead(200, {'Content-Type': 'text/plain'});
        response.end(str);
```

```
    });
}).listen(8080);
```

在代码清单 21.7 中，回调函数并没有被包含在很深的代码层中，因此，改写后也看不出太大差异。然而，如果在之后的代码清单 22.12 那样的代码中进行了类似的改写，代码的执行方式就会变得难以理解。回调函数能够通过闭包机制来实现对外部局部变量的访问，这是一种方便的功能。但如果代码层次很多，则应考虑像代码清单 21.8 这样，重新定义一个新的函数。

代码清单 21.8　改写代码清单 21.7，使其不再使用闭包

```
function callback(response, err, str) {
    if (err) throw err;
    response.writeHead(200, {'Content-Type': 'text/plain'});
    response.end(str);
}

var httpd = require('http').createServer(function(request, response) {
    var filepath = './' + require('url').parse(request.url).pathname;
    require('fs').readFile(filepath, 'utf8', callback.bind(this, response));
}).listen(8080);
```

■ 自定义事件

如果将自定义对象作为事件源，就能对该对象设置事件处理程序。这里需要考虑的是如何才能写出符合 Node.js 风格的代码，除了传统的基于方法的 API 之外，能否使用基于事件的 API 呢？

通常会使用 util.inspect 函数来将自定义对象设定为事件源。事件名可以根据喜好任意选择。不过，'error' 这一名称有特定的用途，所以不应作为事件名使用。代码清单 21.9 介绍了将自定义类设定为事件源的方法。

代码清单 21.9　将自定义类设定为事件源

```
var events = require('events');
var util = require('util');

function MyEventSource() {
    events.EventEmitter.call(this);     // 调用父类的构造函数
}
util.inherits(MyEventSource, events.EventEmitter);   // 继承

MyEventSource.prototype.doit = function(data) {
    // 如有必要，可在此添加 MyEventSource 对象的处理
    this.emit('myevent', data);    // 发出自定义事件。事件名可以使用任意名称
}

// 可以通过下面的代码来使用上面的事件源
var obj = new MyEventSource();
obj.on("myevent", function(data) { // 向自定义事件添加事件处理程序
    console.log('Received data: "' + data + '"');
    });

obj.doit('foobar');
```

如果该对象是一个单例对象，则可以像代码清单 21.10 这样，直接通过 EventEmitter 类来创建对象实例。

代码清单 21.10　将单例对象设定为事件源

```
var EventEmitter = require('events').EventEmitter;
var mysource = new EventEmitter();
mysource.val = 'foobar';
mysource.doit = function(data) {
    this.emit('myevent', data);
};
```

```
mysource.on('myevent', function(data) {
    console.log('Received data: "' + data + '"');
});

mysource.doit('foobar');
```

21.3.10　缓冲

在 JavaScript 以外的程序设计语言中，存在所谓数组的数据结构。通常，数组是一段连续的内存空间，其中存放了特定元素类型的值。

本书第 2 部分提到过，JavaScript 中的数组和其他语言中的数组的性质稍有不同。JavaScript 中的数组是一种对象，只不过其属性名恰好是连续的数值而已。不考虑其内部实现如何，至少它并不能保证使用了连续的内存空间，因此，可能存在效率低下的问题。

在 JavaScript 中也没有字节类型，而在其他很多程序设计语言中，都具有这种类型。于是，在 JavaScript 中也就不存在字节序列及相对应的类了。不过，Node.js 中的 Buffer 类可以被用于弥补这一缺失。

■ Buffer 对象的创建

可以像下面这样，将所需的大小传递给其构造函数，来创建一个 Buffer 类的对象实例。所创建的对象实例，被称为 Buffer 对象。

```
var buf = new Buffer(1024);              // 创建一个大小为 1024 字节的 Buffer 对象
```

在创建之后便无法再更改 Buffer 对象的大小。尽管可以对元素进行改写，但不能超出创建时所定的大小限制，添加更多的元素。

我们还可以通过其他的一些方式来创建 Buffer 对象。比如，可以从其他 Buffer 对象中复制一部分内容，并以此创建新的对象。这种做法相当于使用所谓的复制构造函数来创建对象。还可以通过 JavaScript 基本类型之一的字符串值，或数值数组来创建对象。下面分别是一些具体示例。

```
var buf = new Buffer(buf);               // 相当于通过复制构造函数来创建 Buffer 对象

var buf = new Buffer('foo');             // 通过字符串值来创建 Buffer 对象

var buf = new Buffer([0x61, 0x62, 0x63]);   // 通过数值数组来创建 Buffer 对象（在本例中，其值为 'abc'）
```

■ Buffer 对象与字符串

在 JavaScript 中，字符串的内部编码是 UTF-16（UCS2）。通常情况下，UFT-16 编码的字节序列并没有什么利用价值。因此，在创建字节序列时，一般都会将 UTF-16 编码转换为 UFT-8 编码。在使用字符串值创建 Buffer 对象时，可以通过构造函数的参数来指定字符编码。如果没有指定特定的值，则会将其转换为 UTF-8 编码并创建字节序列。

我们可以通过 toString 方法将 Buffer 对象转换为字符串值，其参数则可用于指定字符编码。能够被指定的字符编码如下所示。

- ascii
- uft8（也可以写作 uft–8）
- base64
- ucs2（也可以写作 ucs–2）

由于在之后的范例中都会使用 utf8 编码，因此，这里仅提供一些使用其他编码的示例。

```
> var buf = new Buffer('abc', 'ascii');
> console.log(buf.toString());
abc
```

```
> var buf = new Buffer('5LiA5LqM5LiJ==', 'base64');
> console.log(buf.toString());
一二三
> console.log(buf.toString('base64'));
5LiA5LqM5LiJ==

> var buf = new Buffer([0x00, 0x4E, 0x8C, 0x4E, 0x09, 0x4E]);
> buf.toString('ucs2');
'一二三'

> var buf = new Buffer('一二三', 'ucs2');
> console.log(buf);
<Buffer 00 4E 8C 4E 09 4E>
```

■ 访问 Buffer 对象中的元素

我们可以通过数值下标记方括号运算符来访问 Buffer 对象中的元素。下标由 0 开始计数（对于大小为 1024 的 Buffer 对象，有效的下标范围为 0~1023）。通过这种方法所获取的元素的值都是数值类型。由于是字节序列，因此这些数值的范围为 0~255。同样地，可以通过方括号运算符改写每一个元素的值。还可以通过 Buffer 对象的 get 与 set 方法来改写其元素。下面是一个具体的例子。

```
> var buf = new Buffer('abc');
> console.log(buf[0]);             // 字符 a 的字节值为 97（=0x61）
97
> console.log(buf.get(0));
97

> buf[0] = 0x41;                   // 将下标为 0 的元素的字节值改写为 0x41（'A'）
> buf.set(1, 0x42);                // 将下标为 1 的元素的字节值改写为 0x42（'B'）

> console.log(buf.toString());     // 将字节序列转换为字符串，得到 'ABc'
ABc
```

以超出范围的下标读取元素值，并不会造成错误，而只是会返回一个 undefined 值。以超出范围的下标改写元素值，同样不会引起错误。这一操作将被忽略。如果没有显式地指定下标，则会获得一个随机值[①]。

可以通过 length 属性获得 Buffer 对象的大小。该大小是以字节为单位的。例如，对于下面这样的日文字符串[②]，其 UTF-8 编码的字节大小为 9。

```
> var buf = new Buffer('あいう');
> console.log(buf.length);         // 字节的个数
9

> var str = 'あいう';
> console.log(str.length);         // 字符的个数
3
```

length 所返回的是其占用的内存大小，而不是实际写入了数据的字节数。例如，下面的例子中，将会返回 32 这一结果。

```
> var buf = new Buffer(32);
> buf.write('abc');
> console.log(buf.length);
32
```

■ Buffer 对象的方法

表 21.8 列出了 Buffer.prototype 对象的属性。关于 prototype 的语言规范，请参见本书第 2 部分。事实上，这些属性就相当于 Buffer 对象所能调用的方法。

---

① 这样的规则和 C 语言很相似，不过今后有可能会被更改。

② 对于中文字符来说同样如此。例如，'一二三'的 UFT-8 编码长度为 9 个字节，而字符个数为 3，所以会得到与示例代码中相同的结果。——译者注

表 21.8　Buffer.prototype 对象的属性

属性名	说明
[n]	访问下标为 n 的元素值
get(n)	返回下标为 n 的元素的字节值
set(n, v)	将下标为 n 的元素的字节值设为 v
write(string, offset=0, encoding='utf8')	将 Buffer 对象的内容写入为字符串 String
toString(encoding='utf8', start=0, end.buffer.lenght)	将字节序列转换为字符串并返回。start 与 end 分别是起始与结束的下标值
slice(start, end)	返回一个 Buffer 对象，其内容是以 start 与 end 为下标值范围的字节序列
copy(targetBuffer, targetStart=0, sourceStart=0, sourceEnd=buffer.length)	返回一个 Buffer 对象，其内容是所指定字节序列的一个副本
inspect()	返回一个字符串值，其内容为 '<Buffer 16 进制数的数列 >'

表 21.9 总结了 Buffer 类的属性。可以以 Buffer.isBuffer(arg) 的方式对其进行使用。

表 21.9　Buffer 类的属性

属性名	说明
isBuffer(obj)	判定参数所接收的对象是否是一个 Buffer 对象。如果是 Buffer 对象，则返回真
字节 Length(string, encoding='utf8')	返回一个以字节为单位的大小，该大小的值是将参数所接收的字符串值转换为字节序列后所得到的

■ Buffer 对象相关的注意事项

由于 Buffer 对象不会进行多余的内存复制操作，因此执行效率很高，但同时在使用时也存在很多风险，必须加以注意。与如今的脚本语言相比，Node.js 中 Buffer 对象的执行方式更接近于 C 语言。

例如，表 21.8 中 slice 方法所返回的 Buffer 对象，将会与原有的对象共享内存空间。因此，如果更改了原有的字节序列，则 slice 所返回的字节序列的内容也会被改变。下面是一个具体的例子。

```
> var buf = new Buffer('abcdef');
> var buf2 = buf.slice(1, 3);        // 基于 buf，取下标范围为 1~3，创建一个新的字符串
> console.log(buf2.toString());      // 得到 'bc'
bc

> buf2[0] = 0x41;                    // 将 buf2 中的第一个字节改写为 'A'
> console.log(buf2.toString());      // 得到 'Ac'
Ac

> console.log(buf.toString());       // buf 所引用的字节序列也会受到影响
aAcdef
```

与 5.12 节中所介绍的不可变对象相反，这是一种危险的代码。在使用时请多加小心。像下面这样，在代码中通过 copy 方法来赋值的话，虽然速度会有所牺牲，但安全性能得到保证。从内部来看，这一方法进行了字节序列的复制操作，因此，更改原字节序列，将不会影响到通过 copy 得到的字节序列。

```
> var buf = new Buffer('abcdef');
> var buf2 = new Buffer(2);

> buf.copy(buf2, 0, 1, 3);           // 基于 buf，取下标范围为 1~3，创建一个新的字符串
> console.log(buf2.toString());      // 得到 'bc'
bc

> buf2[0] = 0x41;                    // 将 buf2 中的第一个字节改写为 'A'
> console.log(buf2.toString());      // 得到 'Ac'
Ac

> console.log(buf.toString());       // buf 所引用的字节序列不会受到影响
abcdef
```

对于了解 C 语言的人来说，可以将这里的 copy 方法理解为使用了 memcpy(2)，而 slice 则仅仅是创建了一个指向同一块内存空间的指针。

21.3.11 流

■ 流的定义

流是一种“可以对数据进行读写操作的功能的抽象”。通常来说，流指的是文件或网络，不过，只要具有可以进行数据读写这一共性，就可以将其视为一种流。通过将功能限定于（即抽象为）数据的读写，就能够对不同的目标使用相同的操作方式。在程序设计中经常会使用到流，通过流的使用，能够切实感受到抽象化的威力。

不光 Node.js，很多的程序设计语言或环境都提供了对流这一抽象功能的支持。不过，在 Node.js 中，流具有异步读写这一特征。Node.js 中的流提供了异步读写功能这样的说法并不准确。事实上，从原则上来说，Node.js 中的流仅支持异步读写操作。仅对异步处理提供支持的设计理念是 Node.js 的一大特征，从这个角度来说，流是 Node.js 中具有代表性的一种功能。

在 Node.js 中，流的功能是通过 Stream 类来实现的。从内部来看，该类继承于 21.3.9 节中所介绍的 EventEmitter 类。不过，开发者并不需要显式地使用 Stream 类。例如，在之后将会说明的网络及文件的读写中，该类将会被隐藏于后台。用 Java 的概念来说明的话，Stream 类相当于一种接口，或抽象类。

■ 流与异步处理

由于在流中仅提供了异步处理，因此 Node.js 中的 Stream 类，与其他程序设计语言及环境中所提供的流类给人的感觉有所不同。在其他的流类中，通常会提供 read 与 write，或着类似于 get 或 put 这样名称的方法。通过调用这些方法，就能读写字节序列或字符串。

在使用 Node.js 的流时，不仅需要调用相应的方法，还要通过回调函数来进行事件处理。也就是说，这一过程中不仅需要调用方法，还需要使用事件及事件处理程序等功能。

■ 读取流

21.10 列出了读取流时所常会用到的事件。

表 21.10 读取流中主要的事件

事件名	事件处理程序	说明
data	function(data) {}	在读取数据时发生
end	function() {}	在读取数据结束时发生
close	function() {}	在流中的文件或套接字关闭时发生
error	function(exception) {}	在出现错误时发生

读取流中最重要的事件就是 data 事件。其事件处理程序的参数将会接收所读取的数据。该参数默认是 Buffer 对象的字节序列。如果事先通过 setEncoding 方法指定了读取流的字符编码，事件处理程序的参数则会接收一个字符串。

表 21.11 列出了读取流中主要的属性。

表 21.11 读取流中主要的属性

属性名	说明
readable	用于标识是否可以读取的旗标。如果发生了错误，则会被置为 false
setEncoding(encoding)	指定字符编码。参见之前的 Buffer 对象与字符串一节
pause()	将 data 事件的发出暂停
resume()	重新开始发出之前通过 pause 暂定的 data 事件
destroy()	关闭（销毁）流内部的文件或套接字
destroySoon()	关闭（销毁）流内部的文件或套接字。不过，这一操作将在缓冲区清空后才开始执行

■ 写入流

在代码中使用写入流时，最基本的操作就是调用 write 方法。表 21.12 中列出了写入流中主要的事

件，表21.13列出了其主要属性。

表21.12　写入流中主要的事件

事件名	事件处理程序	说明
drain	function() {}	在写入缓冲区已满而导致write方法失败后，缓冲区将会被清空。该事件将在缓冲区被清空时发生
close	function() {}	在流内部的文件或套接字被关闭时发生
error	function(exception) {}	在出现错误时发生

表21.13　写入流中主要属性

属性名	说明
writable	用于标识是否可以进行写入的旗标。如果发生了错误，则将会被置为false
write(string, encoding='utf8')	将字符串值写入流中
write(buffer)	将Buffer对象的字节序列写入流中
end()	结束写入
end(string, encoding='utf8')	将字符串值写入流中，并在完成结束写入
end(buffer)	将Buffer对象的字节序列写入流中，并在完成后结束写入
destroy()	关闭（销毁）流内部的文件或套接字
destroySoon()	关闭（销毁）流内部的文件或套接字。不过，这一操作将在缓冲区清空后才开始执行

■ 标准输入输出

在表21.5所列出的process模块属性中，含有一些标准输入输出规格的流对象。对于这些流对象，可以使用本节所说明的方法及事件对其进行操作（代码清单21.11）。

代码清单21.11　以下代码的功能相当于cat（将来自标准输入流的输入流向至标准输出流中）

```
// 默认情况下，标准输入流处于pause状态。为了使其能够接收data事件，必须对其调用resume方法
process.stdin.resume();

// 没有下面两行代码也能正常运行
// 下面的data事件处理程序所传递的参数data是一个字符串值
process.stdin.setEncoding('utf8');
process.stdout.setEncoding('utf8');

process.stdin.on('data', function(data) {
    process.stdout.write(data);
  })
.on('end', function() {
    process.stdin.destroy();
  })
.on('close', function() {
    process.exit();
  })
.on('error', function(ex) {
    process.stdin.destroy();
  });

process.stdout.on('close', function() {
    process.exit();
  })
.on('error', function(ex) {
    process.stdout.destroy();
  });
```

如果使用util模块中的pump函数，则可以将代码清单21.11中的代码简化为以下两行。

```
// 上述代码的简化版本，其功能相当于cat
process.stdin.resume();
require('util').pump(process.stdin, process.stdout);
```

第 22 章 Node.js 程序设计实践

承接前一章的内容，本章介绍 Node.js 程序设计实践。首先介绍以异步方式实现网络及文件处理的实例。之后，将会介绍 Express 及 MongoDB。前者是一种运行于 Node.js 之中的 Web 应用框架，后者是一种面向文档的数据库，可以通过它来进行 Web 应用开发。

22.1 HTTP 服务器处理

代码清单 21.4 已经说明了通过 Node.js 来实现简单的 HTTP 服务器处理的方法。在此我们重新对一些相关术语进行了整理。

由于与 HTTP 相关的 API 被包含于 http 模块之中，因此在使用时需要像代码清单 21.4 那样，载入 http 模块。可以以任意名称的变量来接收 require 函数所返回的对象，不过如果没有特别需求，应该使用 http 这一名称。这样能获得较好的可读性。本节中所有的代码，都将以下面这行语句开始。

```
var http = require('http');
```

在下文中，我们将会对 http 模块中的类添加 http 这一前缀。例如，Server 类将被称为 http.Server。

http 模块中仅提供了基本的 HTTP 功能。如果要开发出真正的 Web 应用，还需要设计一些高层功能。本章之后将说明 Express 等 Web 应用程序框架，可以考虑在开发中使用这些框架。在使用 Java 进行 Web 应用开发时，我们通常会使用框架，而不是直接使用 Servlet API。与之类似地，通常情况下在 Node.js 中也应使用框架来开发 Web 应用。

22.1.1 HTTP 服务器处理的基本流程

http.Server 类是 HTTP 服务器处理中最为核心的一个类。http.Server 对象能够发出表 22.1 中列出的这些事件。

表 22.1 http.Server 的事件

事件名	事件处理程序	说明
request	function(request, response) {}	在收到来自 HTTP 客户端的请求时发生
connection	function(stream) {}	在收到来发 HTTP 客户端的请求时发生（该事件将在 request 事件，以及对请求进行分析处理之前发生）
close	function(errno) {}	在 HTTP 服务器终止时发生
checkContinue	function(request, response) {}	在遇到 Expect:100-continue 这一请求头部时发生
upgrade	function(request, socket, head) {}	在遇到 Upgrade 这一请求头部时发生
clientError	function(exception) {}	在出现网络错误时发生

虽然也可以直接通过 new 来创建 http.Server 实例，不过习惯上，我们会使用 http.createServer 来创建实例。对于 http.Server 类来说，http.createServer 函数相当于一个所谓的工厂函数。http.createServer 函数的参数能够接收一个可选的事件处理程序。它将会被设定为，通过该函数创建的 http.Server 对象的 request 事件的事件处理程序（在对代码清单 21.4 与代码清单 21.5 的比较中也对此进行了说明）。

http.Server 对象具有表 22.2 中列出的这些方法。简单地说，在调用了 listen 方法之后，HTTP 服务器就会开始工作，而在调用了 close 方法之后，运行就会被终止。

表 22.2　http.Server 中主要的方法

方法	说明
listen(port, [hostname], [callback])	通过指定的端口接收 HTTP 连接。通常可以省略 hostname 这一参数。callback 将在连接开始时被调用
close()	中断 HTTP 连接

通过 http.Server 来实现 HTTP 服务器处理的基本流程如下所示。

- **创建 http.Server 对象。**
- **向 http.Server 对象添加必要的事件处理程序（至少要对 request 事件添加事件处理程序，否则 HTTP 服务器无法执行任何功能）。**
- **对 http.Server 对象调用 listen 方法。**
- **借助 request 事件的回调函数，从其参数所接收的 request 对象中抽出请求信息，并通过其参数所接收的 response 对象对响应进行处理。**

在执行了 listen 之后，我们只需接着执行等待事件的代码即可。在显式地调用 close 方法之前，程序都将会等待 HTTP 连接。由于这里执行的是异步处理，即使同时有多个 HTTP 服务器请求连接，也可以对其进行并行处理。不过，由于这里使用的是单线程处理，因此即使存在多个 CPU，也无法实现真正意义上的并行处理。

在从 HTTP 客户端处收到了请求之后，request 事件就会发生。在此之前，将会首先触发 connection 事件。如果读者希望直接对 HTTP 通信协议进行操作，则可以在 connection 事件中进行处理。不过，如果没有特别需求，只要在 request 事件中执行处理即可。事实上，HTTP 服务器处理的核心，就是在 request 事件中对请求及响应进行处理。本章之后将介绍这两种处理。

22.1.2　请求处理

我们可以通过 http.ServerRequest 类来对请求进行处理。可以将 http.ServerRequest 对象传递给表 22.1 中 request 事件的事件处理程序的第 1 个参数。http.ServerRequest 是一个继承于读取流的类。

表 22.3 与表 22.4 分别总结了 http.ServerRequest 对象的事件及其主要属性。

表 22.3　http.ServerRequest 的事件

事件名	事件处理程序	说明
data	function(chunk) {}	在收到 HTTP 请求正文(即所谓的 POST 数据等信息)时发生。参数 chunk 是一个 Buffer 对象或者字符串
end	function() {}	在 1 个 HTTP 请求处理结束时发生
close	function(err) {}	在请求处理中出现网络错误或超时情况时发生

表 22.4　http.ServerRequest 中主要的属性

属性名	说明
url	请求 URL 字符串（包含了查询参数）
method	HTTP 方法字符串（如 'GET' 或 'POST' 等）
headers	HTTP 请求头部所组成的关联数组对象
setEncoding(encoding=null)	如果将字符编码指定为 'utf8'，传递将一个字符串传递给 data 事件的事件处理程序中的 chunk 参数

如果从输入输出的角度来看 HTTP 服务器处理，与输入相对应的就是请求 URL 及其头部信息与正文信息。GET 方法主要通过请求 URL 中的查询参数来接收输入信息，而 POST 方法则主要通过请求正文来获取表单数据。与输出对应的是所返回的响应。

可以通过 headers 属性的值来获取请求头部。该属性是一个关联数组，且以头部名称为键，以头部信息的值为该键所对应的值（从语言规范来看，这是一个普通的 JavaScript 对象）。头部名称全都被进行了标准化，而转换为了小写字母。下面是个具体例子。

```
{ 'user-agent': 'curl/7.18.2 (i486-pc-linux-gnu) libcurl/7.18.2 OpenSSL/0.9.8g
zlib/1.2.3.3 libidn/1.8 libssh2/0.18',
    host: 'localhost:8080',
    accept: '*/*',
    'content-length': '3',
    'content-type': 'application/x-www-form-unlencoded' }
```

■ URL 分析处理

本节之后将以 GET 方法为主，说明分析 URL 并获取其查询参数的值的方法。而对 POST 方法的说明将留到之后章节。我们可以通过 http.ServerRequest 对象的 url 属性来获取 URL 路径及其查询参数的值。url 参数的值是一个形如 /path?foo=bar 的字符串。为了从这一字符串中获得查询参数 foo 的值 bar，必须对其进行分析。

用于对 URL 字符串进行分析的 API 包含于 'url' 模块之中。用于对查询参数进行分析的 API 则包含于 'querystring' 模块中。图 22.1 是一个具体的使用示例。

图 22.1 对 URL 字符串进行分析处理

```
> var url = require('url');

// URL 字符串示例
> var urlstring = 'http://www.example.com/path?foo=bar&baz=abc&baz=xyz';
> console.dir(url.parse(urlstring));     // 通过 url.parse，就能够将 URL 路径与查询参数分离
{ protocol: 'http:',
    slashes: true,
    host: 'www.example.com',
    hostname: 'www.example.com',
    href" 'http://www.example.com/path?foo=bar&baz=abc&baz=xyz',
    search: '?foo=bar&baz=abc&baz=xyz',
    query: 'foo=bar&baz=abc&baz=xyz',
    pathname: '/path' }

> console.dir(url.parse(urlstring, true));
// 将 true 传递至第 2 个参数，就能隐式地对 query 属性的值进行分析（其内部使用了 querystring 模块）
{ protocol: 'http:',
    slashes: true,
    host: 'www.example.com',
    hostname: 'www.example.com',
    href" 'http://www.example.com/path?foo=bar&baz=abc&baz=xyz',
    search: '?foo=bar&baz=abc&baz=xyz',
    query: { foo: 'bar', baz: [ 'abc', 'xyz' ] },
    pathname: '/path' }

> var querystring = require('querystring');
// 对 'foo=bar&baz=abc&baz=xyz' 进行分析
> console.dir(querystring.parse(url.parse(urlstring).query));
{ foo: 'bar', baz: [ 'abc', 'xyz' ] }
```

在实际的 Web 应用中对 request.url 的分析是在 request 事件处理程序中进行的。不过本节开头已经提到过，在实际的开发中，这类处理通常都会被 Web 应用程序框架隐藏。

22.1.3 响应处理

我们可以通过 http.ServerResponse 类来处理响应。可以将 http.ServerResponse 对象传递给表 22.1 中 request 事件的事件处理程序的第 2 个参数。http.ServerResponse 是一个继承于写入流的类。

表 22.5 总结了 http.ServerResponse 所含有的，与构成 HTTP 请求的元素相对应的 5 种方法。

表 22.5　HTTP 响应的元素及 http.ServerResponse 中相应的方法

响应元素	http.ServerResponse 中的方法
状态码	writeHead()。也可以对 statusCode 属性直接赋以数值
响应头部	writeHead()、setHeader()
响应正文	write()、end()

关于 writeHead 方法的使用示例，请参见代码清单 21.4。代码清单 22.1 是 setHeader 方法及响应正文输出的具体示例。

代码清单 22.1 的代码对 write 方法进行了多次调用。在服务器返回响应后，客户端将会接收到一个响应，其响应正文为字符串 foobarbaz[①]。

代码清单 22.1　响应处理

```
var httpd = http.createServer(function(request, response) {
    // 在输出响应正文前，必须对头部信息进行设置
    response.setHeader('Content-Type', 'text/plain');

    // 为了输出响应正文，需要多次对 write 方法进行调用（直至调用了 end 方法为止）
    response.write('foo');
    response.write('bar');
    // 可以将下面的第 2 行省略，只写 response.write('bar'); 一句
    response.write('bar');
    response.end();
}).listen(8080);
```

write 方法及 end 方法的参数可以接受字符串值或 Buffer 对象。如果传递了一个字符串到参数，而需要通过方法的第 2 个参数来设定字符编码。默认情况下字符编码为 'utf8'，因此，通常也可以省略这一参数。如果传递了一个 Buffer 对象，则将会发送其相应的字节序列。

调用 end 方法就意味着要结束响应的输出。如果只是离开了回调函数，并不意味着响应处理已经结束，对此请加以注意。

22.1.4　POST 请求处理

只有在读取了 HTTP 请求正文之后，才能接收 POST 请求中的表单数据。可以通过表 22.3 中的事件来执行异步处理，从而实现这一读取操作。事件处理程序的参数将会接收响应正文的内容。默认情况下，响应正文是一个 Buffer 对象（字节序列）。如果事先通过 http.ServerRequest 对象中的 setEncoding 方法对字符编码进行了指定，则会以字符串的形式传递这一参数。

传递至事件处理程序的正文数据是尚未经过分析的字符串，或者是一个字节序列。需要通过 querystring 模块的函数，将其分析为 HTML 表单中的表单域名与表单域的形式。代码清单 22.2 是个具体的例子。

代码清单 22.2 中代码的功能是分析表单数据，并将其转换为文本数据，之后将该文本作为响应返回。代码清单 22.2 通过 for in 循环枚举了所有的表单域。如果需要获取特定的表单域，则可以使用 formdata['fieldname'] 的形式来对其进行访问。end 事件的作用是检测正文数据的读取是否已经完成，不过，代码清单 22.2 并没有使用该事件。

代码清单 22.2　POST 请求处理

```
var httpd = http.createServer(function(request, response) {
    request.setEncoding('utf8');
    request.on('data', function(chunk) {
// 由于上一行中使用了 setEncoding 方法，chunk 将会是一个字符串
```

① 从 HTTP 层面上来看，请求是以 chunked 的字符编码发送的。然而，HTTP 客户端的操作不应该依赖于所传送响应的字符编码格式。

```
        var querystring = require('querysting');   var formdata = querystring.parse(chunk);
// 将正文数据作为表单的输入值进行分析
        var arr = [];
        for (var key in formdata) {
            arr.push(key + '=' + formdata[key]);    // 通过这种方式，获取表单中的各个表单域值
        }

        response.setHeader('Content-Type', 'text/plain');
        response.end(arr.join(','));
    });

    request.on('end', function() {
        // 请求正文读取完成后的处理
    });
}).listen(8080);
```

22.2 HTTP 客户端处理

可以通过 http.ClientRequest 类与 http.ClientResponse 类，来执行 HTTP 客户端处理[①]。HTTP 客户端处理的基本结构如下所示。

- 创建 http.ClientRequest 对象。
- 向 http.ClientRequest 对象的 response 事件添加事件处理程序。
- 对 http.ClientRequest 对象进行请求写入操作（必须调用 end 方法）。
- response 事件的事件处理程序将会接收一个 http.ClientResponse 对象，可以从该对象中获取响应信息。

HTTP 客户端处理的内容仅有在发出请求后等待响应而已。可以通过 write 方法与 end 方法来对请求正文进行写入操作。如果调用了 end 方法，就表示请求处理已经完成。即使没有请求正文，也需要调用 end 方法。如果没有调用 end 方法，请求处理就不会结束，对此请加以注意。而等待响应的操作则一如既往地，是借助于事件来实现的，这也符合 Node.js 风格。这时，要使用的事件是 http.ClientRequest 对象中的 response 事件。

代码清单 22.3 是一个 HTTP 客户端处理的代码示例。虽然也能通过 new 来创建 http.ClientRequest 对象，不过，我们习惯通过 http.request 函数来创建该对象。http.request 函数的第 1 个参数将会接收所连接的服务器的信息，而其第 2 个参数则会接收一个回调函数。传递给这一参数的函数将会被作为 http.ClientRequest 对象中 response 事件的事件处理程序。代码清单 22.3 没有使用 http.request 函数的第 2 个参数，而是显式地使用了 on 函数。

代码清单 22.3 Node.js 中的 HTTP 客户端

```
var http = require('http');

// 所连接的服务器的信息
var options = {
    host: 'www.google.com',
    port: 80,
    path: '/',
    method: 'GET'
};

// 返回值是一个 http.ClientRequest 对象
var req = http.request(options);
```

① http.Client 类是一个较旧的 API。之所以现在它仍然存在，仅仅是出于兼容性的考虑。今后请大家使用 http.ClientRequest 和 http.ClientResponse。

```
// 这一回调函数将在收到响应时被调用（也可以将其传递至上面的 http.request 函数的第 2 个参数）
req.on('response', function(res) { // 传递给事件处理程序的参数 res，是一个 http.ClientResponse 对象
    // 在调用了 setEncoding 之后，下面的 chunk 将是一个字符串（如果没有调用，则会是一个 Buffer 对象）
    res.setEncoding('utf8');

    // 读取响应正文
    res.on('data', function(chunk) {
        console.log('BODY.' + chunk);
    });
});

// 如果发生了错误，则会调用该事件处理程序
req.on('error', function(e) {
    console.log('error: ' + e.message);
});

// 如果响应中包含有正文（如 POST 处理等），则通过 write 对正文进行写入。可以多次调用 write
// req.write('request body data');

// 调用了 end 方法之后，就表示请求已经被发送。即使没有使用 write 方法，也必须调用 end 方法
req.end();
```

response 事件的事件处理程序将会接收一个 http.ClientResponse 对象作为参数。可以从该对象中获取响应的正文信息。如代码清单 22.3 所示，我们可以通过 data 事件，以异步处理的方式读取正文信息。data 事件的事件处理程序的参数将会接收响应的正文数据。如果实现通过 setEncoding 方法指定了字符编码，正文数据则会被转换为字符串型，否则将以 Buffer 对象（字节序列）的形式获取响应正文。

http.get() 是一种特殊的 GET 方法，它是一个在 GET 的基础上简化而成的 API。且该 API 在其内部隐式地执行了类似于 req.end() 的操作，使用起来很容易。由于较为简单，这里就不再对其使用方法进行说明。

22.3 HTTPS 处理

接下来，我将介绍 HTTPS 处理。为了运行一个 HTTPS 服务器，需要做一些准备工作。由于本书并非专门介绍 HTTPS 的图书，因此本节仅说明使用自签名证书搭建简易 HTTPS 服务器的方法。

22.3.1 通过 openssl 指令发布自签名证书的方法

本节将介绍通过 openssl 指令发布自签名证书的方法。只需执行以下代码即可。

```
// 通过 openssl 指令发布自签名证书
$ openssl req -new -x509 -keyout key.pem -out cert.pem
```

在执行了这一代码之后，系统将会以会话的形式询问私钥的口令短语和证书的可分辨名称（DN），需要按要求输入这些信息。完成后，将会获得 2 个输出文件，其中 key.pem 是私钥文件，而 cert.pem 则是证书文件。这两个文件的文件名是可以任意选择的，不过，由于之后将在 Node.js 代码中对其进行引用，因此需要确保此处的名称与代码中所使用的名称相一致。

22.3.2 HTTPS 服务器

在代码清单 22.4 的代码示例中，我们通过所生成的私钥和证书创建了一个 HTTPS 服务器。其中，使用了 8443 号端口。在启动服务器之前，需要将私钥与证书这两个文件存放于相应路径。

代码清单 22.4　HTTPS 服务器

```
var fs = require('fs');
```

```
var options = {
    key: fs.readFileSync('key.pem'),
    cert: fs.readFileSync('cert.pem')
};

var httpsd = require('https').createServer(options, function(request, response) {
    response.writeHead(200, {'Content-Type': 'text/plain'});
    response.end('Hello World\n');
}).listen(8443);
```

可以像下面这样，通过 curl 指令确认其运行情况。在使用了 -k 选项后，即使不指定可分辨名称，也不会发生错误。

```
$ curl -k https://localhost:8443/

// 或者，可以将通过 openssl 指令所生成的 cert.pem 存放于相应路径，并执行以下指令
//（这时，如果可分辨名称等信息不正确，则会发生错误）
$ curl --cacert cert.pem https://localhost:8443/
```

只要将代码清单 22.3 中的 require('http') 替换为 require('https')，即可得到 HTTPS 客户端的代码示例，因此不再赘述。

22.4 Socket.IO 与 WebSocket

本书第 4 部分介绍了通过 Node.js 实现了 WebSocket 的代码示例。第 4 部分所使用的是 websocket-server 包，本节将会介绍一个通过 Socket.IO 包来实现该功能的例子。

Socket.IO 并不是一个专用于 WebSocket 的包，WebSocket 是该包所提供的基础技术之一，通过该包，能够实现各种类型的 Web 实时通信功能。如果运行环境不支持 WebSocket，可以使用第 17 章介绍的长轮询等替代方式，来实现 Web 实时通信。此时，无需考虑这些方法使用了哪些底层技术。

我们可以在下面的站点中获取 Socket.IO 包的相关信息。

http://socket.io/

我们可以通过下面的 npm 指令来安装 Socket.IO 包。

```
$ npm install socket.io
```

在 Socket.IO 包中，集成了针对 Node.js 所使用的服务器端 JavaScript 库，及运行于浏览器中的客户端 JavaScript 库。于是，在使用该包时，我们能够在服务器端和客户端同时使用 JavaScript 这一程序设计语言。得益于此，我们可以在开发中使用（几乎）相同的 API。

下面是在服务器端（Node.js）中使用 Socket.IO 包的基本方式。

- 调用 Socket 对象的 listen 方法。
- 通过 io.sockets 的 connection 事件，获取一个已经与客户端建立了连接的 io.Socket 对象。
- 通过 io.socket 的 emit 方法来发送消息（可以指定任意的事件名）。
- 通过 io.socket 的事件来接收消息。

在客户端（浏览器）JavaScript 中使用 Socket.IO 时，需要在代码中通过 <script src="socket.io.js"></script> 来读取 Socket.IO 所提供的客户端文件。其本使用方式如下所示。

- 通过 io.connect('服务器 URL') 来创建 io.Socket 对象。
- 通过 io.socket 的 emit 方法来发送消息（可以指定任意的事件名）。
- 通过 io.socket 的事件来接收消息。

下面的代码是以上两种方式的简单示例（代码清单 22.5 与代码清单 22.6）。服务器端与客户端将以 3 秒为间隔进行消息收发操作。emit 方法的第 1 个参数可以使用任意的事件名。其第 2 个参数可以接收任意的对象或值（数值及字符串值）。一旦建立了联机，两者的结构就是对称的。

代码清单 22.5　在服务器端使用 Socket.IO

```
var io = require('socket.io').listen(8080);

io.sockets.on('connection', function(socket) {
    // 取事件名为 'msg'，以 3 秒一次的频率向客户端发送对象
    setInterval(function() {
        socket.emit('msg', { msg:'from server', now: new Date() });
    }, 3000);

    // 接收来自客户端的 'msg' 事件
    socket.on('msg', function(data) {
        console.log('from client ', data);
    });

    socket.on('disconnect', function() {
        console.log('disconn');
    });
});
```

代码清单 22.6　在客户端使用 Socket.IO

```
<script src="socket.io.js"></script>
    var socket = io.connect('http://localhost:8080');

    // 接收来自服务器端的 'msg' 事件
    socket.on('msg', function(data) {
        console.log(data);
    });

    // 取事件名为 'msg'，以 3 秒一次的频率向服务器端发送对象
    setInterval(function() {
        socket.emit('msg', { msg:'from client', now: new Date() });
    }, 3000);
```

22.5 下层网络程序设计

22.5.1　下层网络处理

如表 22.6 所示，Node.js 提供了一些用于执行下层网络处理的模块。这些模块将被用于进行所谓的套接字程序设计。

如果采用的是 HTTP 或 SMTP 等具有明确规则的通信协议，使用专用模块将更为方便。而套接字层程序设计将会使用这些专用模块内部的 API。在设计新的通信协议时，我们也会进行套接字级别的程序设计。在这一层级中，将仅执行字节序列的收发操作。这时，上层通信协议将会对所收发的字节序列的内容及发送顺序的含义进行解释。

表 22.6　下层网络模块

模块名	说明
net	用于 TCP 套接字程序设计
dgram	用于 UDP 套接字程序设计
dns	用于域名解析（DNS）

本书将说明 net 模块的使用方法，而对 dgram 及 dns 模块的说明则将省略。关于它们的详细内容，请参见各自的 API 参考文档。

22.5.2 套接字的定义

在传统的网络数据收发中，使用了一种名为套接字的抽象层。试想，在通信双方间有一条一一对应的虚拟连线，在这条线的两端，将对数据进行读写操作。人们对这两个端点的功能进行抽象，并将其命名为套接字。21.3.11 节已经介绍了专用于数据读写的流。套接字正是一种流。本节将仅对 TCP 套接字进行说明。UDP 套接字的原理与其稍有不同，在使用时需要加以注意。

对套接字流进行数据读写操作是下层网络程序设计的基本内容。只要在通信中从套接字流中读取数据，就能够获取对方所发送的数据。而在通信中将数据写入套接字流之后，该数据就会被发送至对方。不同于文件流与内存流，对于套接字流来说，对方将会根据情况决定是否读取所接收的数据。如果是数据接收方，是否能够收到数据也是由对方决定的。

■ 套接字的类型

在使用套接字之前，首先需要与通信对象建立连接。而要识别连接对象，则先要获得对方的 IP 地址。在识别连接对象时，还需要用到端口号。可以将 IP 地址直观地理解为识别对主机进行识别的依据，而将端口号理解为对主机中特定进程进行指定的依据。

在网络通信中常常会使用服务器与客户端这样的名称来区分通信双方。而对于套接字来说，并不会区分服务器与客户端。在这种情况下，仅存在等待连接的套接字（被动套接字）与主动连接的套接字（主动套接字）这两种分类。通常，服务器将会使用被动套接字，而客户端则会使用主动套接字。

在创建被动套接字时，需要指定等待端口。在创建主动套接字时，需要指定连接对象的 IP 地址及端口。主动套接字自身的端口号通常会由系统任意指定，通常，系统会选择一个空闲端口作为套接字的端口。

■ 套接字的执行方式

主动套接字将会尝试与正在等待的被动套接字连接。如果等待中的套接字接受了连接请求，则会自动创建一个接收套接字。

在建立了连接之后，就能通过套接字在通信双方间相互进行数据读写操作。实际进行这一数据通信过程的，是接收套接字（服务器端）和主动套接字（客户端）。之前等待的连接套接字将不会参与通信，对此请加以注意。

虽然接收套接字和主动套接字在名称和创建方式上各不相同，但其执行方式却是对称的。换句话说，在连接开始之前，两者的执行方式是不对称的，而在建立了连接之后，则不存在是由哪一方发起连接的区别。一旦连接被建立，2 个套接字之间的关系是对称的，双方都可以进行数据发送。同时，双方也都可以通过套接字的关闭方法来结束通信。

需要注意的是，之前等待的套接字将会就此保留。如果有来自其他客户端的连接请求，则会继续创建新的接受套接字。如此一来，具有被动套接字的计算机（通常是服务器进程）可以同时接受来自多个（客户端）计算机的连接。

22.5.3 套接字程序设计的基本结构

服务器端代码的基本结构如下所示。

- 调用 net.createServer()，通过其返回值获取 net.Server 对象（将事件处理程序 connection 传递至 createServer 的参数）。

- 调用 net.Server 对象的 listen 方法，进入连接等待状态。
- 通过 net.Server 对象的 connection 事件处理程序的参数，获取与客户端建立了连接的接收套接字（net.Socket 对象）。
- 通过 write 方法向 net.Socket 对象写入数据，并通过 data 事件执行读取操作（每个客户端都需要进行套接字操作）。
- 通过 net.Socket 对象的 close 事件或 error 事件来结束与每一个客户端的连接。
- 调用 net.Socket 对象的 close 方法来切断与服务器的连接。
- 调用 net.Server 对象的 close 方法，结束服务器 2。

上一节已经介绍过，net.Server 类相当于等待连接的套接字。在系统调用层或 POSIX API 层中等待连接的套接字和用于数据收发的套接字是同一种套接字。不过，将前者看作具有另一种功能的套接字（创建用于套接字流的套接字工厂）更好理解。因此，将其抽象为 net.Server 类是一种恰当的设计。net.Server 类起到了接收套接字的容器的功能。

通过 connection 事件，可以得到接受套接字对象。21.3.11 节已经介绍了流的基本读写操作。可以通过 write 方法对套接字流进行写入操作，通过 data 事件读取其数据。

客户端代码的基本结构如下所示。

- 调用 new.createConnection()，以获取 net.Socket 对象（主动套接字）。
- 调用 net.Socket 对象的 connect 方法，向参数所指定的服务器发起连接。
- 通过 net.Socket 对象的 connect 事件来判断是否成功连接。
- 通过 write 方法向 net.Socket 对象写入数据，并通过 data 事件执行读取操作。
- 通过 net.Socket 对象的 close 事件或 error 事件来结束与每一个服务器的连接。
- 对 net.Server 对象调用 close 方法，以切断与服务器的连接。

一旦连接建立，就会触发 connect 事件。对主动套接字进行读写操作的方法与服务器中的情况相同。

正如前一节所介绍的，连接一旦被建立，服务器端与客户端将会对 net.Socket 对象使用相同的读写操作。这里并不是笼统地将两种不同的操作视为了相同操作。之所以说两者是相同的，是因为从本质上来看，该套接字的执行方式是对称的。虽然等待连接的（服务器端）套接字和进行连接的（客户端）套接字在最初是非对称的，不过只要建立了连接，两者就不再有任何差别了。双方都可以通过套接字发送数据，或者对所收到的数据进行读取。从程序的角度来看，它们都是可供读写的流。

表 22.7 列出了 net.Socket 类的一些属性。表 22.8 列出了 net.Socket 类中主要的事件。

表 22.7　net.Socket 的属性

属性名	说明
connect(port, [host], [callback])	连接指定的服务器
setEncoding(encoding)	如果指定了字符编码，传递给 data 事件处理程序的参数则会被转换为字符串
write(data, [encoding], [callback])	将字符串值或 Buffer 对象（字节序列）写入套接字
end([data], [encoding])	将字符串值或 Buffer 对象（字节序列）写入套接字，并在完成后结束写入操作
destroy()	关闭套接字（销毁）
pause()	暂停发出 data 事件
resume()	继续发出之前通过 pause 暂停发出的 data 事件
setTimeout(timeout, [callback])	对套接字设定超时值（单位为毫秒）。如果超时，将会发出 timeout 事件。可以通过第 2 个参数来接收 timeout 事件处理程序
address()	返回套接字的本地地址

表 22.8 new.Socket 中的主要事件

事件名	事件处理程序	说明
connect	function() {}	在套接字连接被建立后发生（用于主动套接字中）
data	function(data) {}	在读取数据时发生。事件处理程序的 data 参数可以是字符串或 Buffer 对象
end	function() {}	在数据读取完成时发生
timeout	function() {}	在超时时发生（默认情况下不会发生。只有在通过 setTimeout 方法显式地指定了超时值之后才会发生）
drain	function() {}	在写入缓冲区已满而导致 write 方法失败后，缓冲区将会被清空。该事件将在缓冲区被清空时发生
error	function(exception) {}	在出现错误时发生
close	function(had_error) {}	在套接字被关闭时发生

22.5.4 套接字程序设计的具体实例

上一节介绍了套接字通信中代码基本结构。本节将会介绍一些具体的代码示例（代码清单 22.7 与代码清单 22.8）。

在服务器端的代码（代码清单 22.7）中，我们选择 9000 号端口等待来自客户端的连接。当收到来自客户端的连接请求时，将会发生 connection 事件。可以通过事件处理程序的参数来获取接收套接字对象。从该对象的 data 事件中可以获取所接收的数据。在以标准输出格式将所读取的数据输出至 console.log 之后，将会通过 sock.write 向客户端输出消息。

在客户端的代码（代码清单 22.8）中，将会连接 localhost 的 9000 号端口。如果服务器进程运行于其他的计算机中，则需要通过 createConnection 的参数来指定远程 IP 地址。在建立连接后，将会发生 connect 事件。之后，可以通过 write 方法发送数据，并通过 data 事件等待回复。

在下面的代码中，客户端将会发送一条消息，之后，服务器将会据此返回一条消息。由于已经事先决定了处理方式，因此可以随意地设计通信协议。此外，在这一代码中，如果发生超时，就将会切断通信。而在实际的应用中，协议的设计者还需要考虑如何以恰当的方式结束通信。

可以通过 telnet 或 nc 指令来确认客户端的执行情况。请确认服务器是否能够同时连接多个客户端，以正确执行并行通信。由于此时采用的是异步网络处理，因此服务器端不使用多线程，也能够实现与多个客户端的连接。

代码清单 22.7 服务器端的代码示例

```
var net = require('net');

var server = net.createServer();
server.listen(9000);

server.on('connection', function(sock) {
    sock.setEncoding('utf8');
    sock.setTimeout(3000);                            // 3 秒

    sock.on('data', function(data) {
        console.log('Via client: [', data, ']');     // 显示所接收的数据
        sock.write('hello from server');              // 在收到数据后，再发送一些数据
    })
    .on('end', function() {
        sock.destroy();
    })
    .on('error', function() {
        sock.destroy();
    })
    .on('timeout', function() {
        sock.destroy();
    })
    .on('close', function() {
        console.log('closed');
```

```
    });
});
```

代码清单 22.8　客户端的代码示例

```
var net = require('net');

var sock = net.createConnection(9000);

sock.on('connect', function() {
    sock.setEncoding('utf8');
    sock.setTimeout(3000);                          // 3 秒
    sock.write('hello from client');
    })
    .on('data', function(data) {
        console.log('Via server: [', data, ']');    // 显示所接收的数据
    })
    .on('end', function() {
        sock.destroy();
    })
    .on('error', function() {
        sock.destroy();
    })
    .on('timeout', function() {
        sock.destroy();
    })
    .on('close', function() {
        process.exit();
    });
```

22.6　文件处理

在 Node.js 中，我们可以通过 fs 模块来执行文件处理，并通过 path 模块来执行与文件路径相关的处理。文件处理函数分为同步和异步两种版本。对于异步版函数，可以接收一个回调函数作为参数（需要将其传递至函数的最后一个参数）。在处理完成时，将会调用该回调函数。该回调函数的第一个参数是一个错误参数。如果错误参数为真，则表示出现了某种错误。

直到前一节为止，Node.js 中的网络函数都是非阻塞式的异步函数。网络处理的本质决定了它可能会花费大量时间，因此，对于必须以事件驱动方式执行的 Node.js 来说，这些非阻塞式操作是不可缺少的。而另一方面，在对文件进行操作时，读取操作只需花费很少的时间。至少在读取本地文件时情况如此。因此，即使将事件驱动方式与同步文件读取操作相结合，也不会对响应性能产生多少影响[①]。

22.6.1　本节的范例代码

本节所有的范例代码之前，都需要首先执行代码清单 22.9 中的代码。并且，这里假定所有的目标文件路径都已经通过命令行参数传递给了代码。

代码清单 22.9　进行文件处理时所需执行的通用操作

```
var fs = require('fs');
var fpath = process.argv[2];        // 命令行参数
if (!fpath) {
    process.exit();
}
```

① 不过，知道磁盘读取的速度很慢的读者，或许不赞同这一观点。之所以这里称不会对性能产生影响，是因为虽然可以将文件 API 视为一种同步 API，但从操作系统（内核）的层面来看，它们仍然是以异步的方式执行的。

我们可以通过命令行参数，将指定的文件名传递给变量 fpath。例如，如果将代码清单 22.9 中的代码保存于文件 my.js 中，并像下面这样执行 node 指令，fpath 的值就将被赋为 'foo.txt'。

```
$ node my.js foo.txt
```

22.6.2　文件的异步处理

下面是以异步方式调用文件的 stat 函数的例子（代码清单 22.10）。stat 函数的参数将会接收一个文件路径，并返回以此为路径的文件的信息。不过，由于这是一种异步处理，stat 函数并不会直接返回文件的信息，而会以回调函数参数的形式返回。

代码清单 22.10　以异步方式调用文件的 stat 函数

```
fs.stat(fpath, function(err, stats) {   // 通过参数来指定回调函数
    if (err) throw err;
    console.log('stats: ', stats);      // 通过回调函数的 stats 参数来获取文件信息
});
```

执行以上代码，将会得到下面的结果。

```
$ node stat.js /tmp
stats: { dev: 2050,
    ino: 895841,
    mode: 17407,
    nlink: 13,
    uid: 0,
    gid: 0,
    rdev: 0,
    size: 12288,
    blksize: 4096,
    blocks: 24,
    atime: Sun, 26 Jun 2011 15:11:54 GMT,
    mtime: Sun, 26 Jun 2011 14:38:51 GMT,
    ctime: Sun, 26 Jun 2011 14:38:51 GMT }
```

回调函数的第 1 个参数将会接收错误状态，而其第 2 个参数则会接收一个 stat 对象（stat 结构体）。可以从上面的执行结果中，判断 stat 对象的属性是否表示 man 2 stat。size 属性在实际使用中非常重要。该属性表示目标文件的大小，将以字节为单位。

与 stat 函数类似的还有 fstat 与 lstat 函数。fstat 的参数接收的不是文件路径，而是文件描述符。该函数将在之后的文件读取范例中使用。lstat 与 stat 几乎相同，不同的是，当它的文件路径参数是一个符号链接时，不会查看链接目标。stat 函数将会返回符号链接的链接所指定的文件的信息。而 lstat 函数则会返回符号链接文件的信息。

22.6.3　文件的同步处理

代码清单 22.11 是代码清单 22.10 所示 API 的同步版。

代码清单 22.11　代码清单 22.10 相对应的同步版

```
var stats = fs.statSync(fpath);
console.log('stats: ', stats);
```

上述代码的执行结果与代码清单 22.10 相同。statSync 函数的返回值是一个 stat 对象。为了与代码清单 22.10 对应，我们可以通过与代码清单 22.10 相同的变量名来接收该对象。如果指定路径的文件不存在，或者发生了其他一些错误，statSync 函数将会抛出一个异常。之前说过，stat 函数不会直接抛出异常，而会将该错误返回给回调函数的第 1 个参数。在这一点上两者的执行方式是不同的。

22.6.4　文件操作相关函数

表 22.9 分别列出了一部分文件操作相关函数的同步版和异步版。其中，异步函数仅比同步函数多了一个回调函数参数。异步函数的回调函数将会接收 1 个表示错误状态的参数。

表 22.9　文件操作类函数（摘选）

异步函数	同步函数	说明
rename(oldpath, newpath, callback)	renameSync	重命名文件
truncate(fd, len, callback)	truncateSync	截取文件。如果 len 为 0，则会清空文件
chmod(path, mode, callback)	chmodSync	更改文件的权限位。参数 mode 的含义请参见下文
unlink(path, callback)	unlinkSync	删除文件

我们首先说明一下 chmod 函数所接收的文件权限位（mode）。在之后将介绍的 open 中也会用到该参数。这一权限位的标准遵循了 Unix 系操作系统的文件系统传统规范。通过位信息，可以设定文件的所有用户、所有组，及其他权限所有者的读取、写入、执行权限。例如，对于 u+rwx（用户具有所有权限）、g+rx（组具有读取及执行权限）、a+rx（其他用户具有读取及执行权限），依次将其转换为比特位 1/0 的形式，再转为 8 进制数，就能得到 0755 这一结果。可以将 0755 这个值指定为 mode 的值。

这一 mode 值存在两个问题。其一是 8 进制数字面量表示问题。ECMAScript 第 5 版不支持从 0 开始的 8 进制数字面量表示（参见第 2 部分中的 3.4.1 节）。虽然也可以通过 10 进制数或 16 进制数来表示权限位，不过这样一来，可读性就会下降。另一个问题是可移植性问题。目前来看，mode 值非常依赖于 Unix 系操作系统。在将这类 API 移植到其他的操作系统时，它将对系统有着过强的依赖性。随着今后 Node.js 所支持的平台越来越多，这类 API 的抽象程度可能会逐步提升，这一问题应该就能有所缓解。

22.6.5　文件读取

在 Node.js 中，通用文件读取处理也采用了异步处理的方式。从 open 至实际的读取操作都是如此（代码清单 22.12）。

在代码清单 22.12 中，open 方法的参数 'r' 表示只读模式。也可以用 require('constants').O_RDONLY 替换 'r'，两者的效果相同。

代码清单 22.12　整个文件读取操作都采用了异步处理方式

```
fs.open(fpath, 'r', function(err, fd) {
    if (err) throw err;

    fs.stat(fpath, function(err, stats) {
        if (err) throw err;

        fs.read(fd, new Buffer(stats.size), 0, stats.size, null,
                function(err, bytesRead, buf) {
                    if (err) throw err;

                    if (!bytesRead) return;
                    console.log(buf.toString());
                });
    });
});
```

异步处理存在代码可读性较低的缺点。在执行文件读取操作时，结合使用同步处理的方式也是一种可选项（代码清单 22.13 与代码清单 22.14）。如果是本地文件，读取时间几乎可以忽略。这时，应该优先考虑代码的简洁。

非要追求代码的可扩放性的话，则应遵循 Node.js 的规范，全部使用异步处理。不过这种做法并不能保证最快的响应速度。书写代码时应该优先考虑哪一方面，是由系统的设计思路所决定的，不需要过分

拘泥于设计出唯一正确的方式。

代码清单 22.13 open 是异步处理，而读取操作是同步处理

```
fs.open(fpath, 'r', function(err, fd) {
    if (err) throw err;

    fs.stat(fpath, function(err, stats) {
        if (err) throw err;

        var buf = new Buffer(stats.size);
        var bytesRead = fs.readSync(fd, buf, 0, stats.size, null);
        if (!bytesRead) return;
        console.log(buf.toString());
    });
});
```

代码清单 22.14 全部使用同步处理

```
var fd = fs.openSync(fpath, 'r');
var buf = new Buffer(4096);
while (true) {
    var bytesRead = fs.readSync(fd, buf, 0, buf.length, null);
    if (!bytesRead) break;
    console.log(buf.toString('utf8', 0, bytesRead));
}
fs.closeSync(fd);     // 如果没有调用 close 或 closeSync，则会发生资源泄漏，对此请加以注意
```

还存在一些简单的文件读取 API。下面是 readFile 和 readFileSync 的代码示例，其中 readFile 是一个异步 API，而 readFileSync 则是一个同步 API（代码清单 22.15 与代码清单 22.16）。

代码清单 22.15 使用 readFile 的版本

```
// 直接读取字节序列
fs.readFile(fpath, function(err, buf) {
    if (err) throw err;
    console.log(buf.toString());    // Buffer 类型
});

// 指定字符编码，以字符串形式进行读取操作
fs.readFile(fpath, 'utf8', function(err, str) {
    if (err) throw err;
    console.log(str);               // 字符串型
});
```

代码清单 22.16 使用 readFileSync 的版本

```
// 直接读取字节序列
var buf = fs.readFileSync(fpath);
console.log(buf.toString());

// 指定字符编码，以字符串形式进行读取操作
var str = fs.readFileSync(fpath, 'utf8');
console.log(str);
```

22.6.6 文件写入

本节将介绍一下异步文件写入操作的代码示例（代码清单 22.17）。需要注意的是，在对流调用 write 方法时，可能会发生异常，因此，通常需要考虑例异常处理的问题。不过，代码清单 22.17 并没有对此进行处理。

代码清单 22.17 异步文件写入操作

```
var buf = new Buffer('data to write');
```

```
fs.open(fpath, 'w', function(err, fd) {
    if (err) throw err;

    fs.write(fd, buf, 0, buf.length, null,
        function(err, bytesWritten, wbuf) {
            if (err) throw err;
        });
});
```

代码清单 22.18 是同步文件写入操作的代码示例。

代码清单 22.18　同步文件写入操作

```
var buf = new Buffer('data to write');

var fd = fs.openSync(fpath, 'w');
var bytesWritten = fs.writeSync(fd, buf, 0, buf.length, null);
fs.closeSync(fd);
```

还存在一些简单的文件写入 API。下面是 writeFile 和 writeFileSync 的代码示例，其中 writeFile 是一个异步 API，而 writeFileSync 则是一个同步 API（代码清单 22.19）。

代码清单 22.19　简单的文件写入 API

```
// 异步 API
fs.writeFile(fpath, buf, function(err) {
    if (err) throw err;
});

// 同步 API
fs.writeFileSync(fpath, buf);
```

21.3.11 节介绍了一段与 cat 拥有相同功能的代码。在那段代码中，使用了 util 模块的 pump 函数。可以通过文件名，来得到相应的读取流和写入流，因此，只需像下面这样使用一行代码，就能够实现复制文件的效果。

```
// 复制文件，并将其保存为扩展名为 bak 的新文件
require('util').pump(fs.createReadStream(fpath), fs.createWriteStream(fpath + '.bak'));
```

22.6.7　目录操作

表 22.10 总结了目录操作相关的 API。

表 22.10　目录操作相关的 API

函数名（异步）	同步	说明
mkdir(path, mode, callback)	mkdirSync	创建目录。关于参数 mode，请参见文件操作相关函数一节中的说明
rmdir(path, callback)	rmdirSync	删除目录
readdir(path, callback)	readdirSync	读取目录一览表

传递给 mkdir 与 rmdir 的回调函数只会接收 1 个出错参数。而传递给 readdir 的回调函数，则会接收出错参数及文件一览表参数这两个参数。文件一览表参数是一个由文件名字符串组成的数组。代码清单 22.20 是该函数的使用示例。

代码清单 22.20　读取目录一览表

```
fs.readdir(fpath, function(err, files) {     // files 是由文件名组成的数组
    if (err) throw err;
    files.forEach(function(filename) {
        console.log(filename);
    });
});
```

22.6.8 对文件更改的监视

可以通过表 22.11 中的函数监视文件的更改。watchFile 函数的功能是开始文件监视，而 unwatchFile 则能停止文件监视。

表 22.11 能够监视文件更改的 API

函数名	说明
watchFile(filename, [options], handler)	当指定文件被更改时，将会调用回调函数
unwatchFile(filename)	停止对指定文件的监视

watchFile 最后的参数将会接收一个回调函数。一旦文件发生了更改，该回调函数就将被调用。即使所传递的文件路径不存在文件，依然可以对其进行监视。这时，将会在该路径下的文件被创建时调用回调函数。回调函数将会接收两个参数。第 1 个参数是当前的 stat 对象，第 2 个参数是文件被更改前的 stat 对象。代码清单 22.21 是一个具体的例子。

代码清单 22.21 watchFile 函数的使用示例

```
fs.watchFile(fpath, function(curr, prev) {
    console.log('the current mtime is: ' + curr.mtime);
    console.log('the previous mtime was: ' + prev.mtime);
});
```

22.6.9 文件路径

path 模块包含一些能够对文件路径进行处理的 API。表 22.12 列出了其中一些具有代表性的函数。其中，除了 exists 相关的函数外，都是字符串处理相关的 API。由于它们不需要执行 I/O 处理，因此没有被设计为异步 API 的形式。

表 22.12 path 模块的函数（摘选）

函数名	说明
dirname(path)	返回参数所指定的文件路径中，除去文件名以外的部分
basename(path, ext)	返回参数所指定的文件路径的文件名。若其中包含扩展名 ext，则将去除该扩展名
extname(path)	返回参数所指定的文件路径下的文件扩展名
exists(path, callback)	检查参数所指定的文件路径下是否存在文件，并调用回调函数。该回调函数具有一个参数，如果文件存在，则其值为 true，否则为 false
existsSync(path)	如果参数所指定的文件路径下存在文件，则返回 true，否则返回 false

下面是一个 exists 函数的使用示例（代码清单 22.22）

代码清单 22.22 path.exists 函数的使用示例

```
require('path').exists(fpath, function(ret) {
    console.log(ret ? "%s exists" : "%s doesn't exist", fpath);
});
```

22.7 定时器

Node.js 对表 22.13 列出的定时器函数提供了支持。这些函数与客户端（DOM）中相应函数的形式相同。从内部来看，它们都是 timers 模块的函数，不过由于默认情况下，该模块将会被载入，因此在使用前不需要显式地使用 require 函数。

表 22.13 定时器函数

函数	说明
setTimeout(callback, delay, [arg…])	在 delay 毫秒后，以参数 arg 调用回调函数。其返回值为 timerId
clearTimeout(timerId)	解除由 timerId 所指定的定时器回调函数
setInterval(callback, delay, [arg…])	以 delay 毫秒为频率，以参数 arg 调用回调函数。返回值为 intervalId
clearInteval(intervalId)	解除由 intervalId 所指定的间隔定时器回调函数

回调函数中的 this 引用指向的是全局对象。如果希望其引用其他的对象，则需要使用 bind 方法（参见事件处理程序内的 this 引用一节的内容）。

专栏

在 Node.js 中进行调试

我们可以像下面这样，以会话的形式启动调试程序。

```
$ node debug my.js
debug>
```

如果读者希望了解能够在调试程序中使用哪些指令，可以通过键入 help 来获取指令列表。在调试状态下，程序将会在的代码中写有 debugger 语句之处中断执行。ECMAScript 第 5 版制定了 debugger 语句的标准（参见本书第 2 部分）。

22.8 Express

Express 是一种用于 Web 应用开发的 MVC 框架，它基于同一作者所开发的 Connect 发展而成。可以通过下面的 URL 获取相关信息。

http://expressjs.com/

可以像下面这样，来安装 Express。

```
$ npm install express
```

代码清单 22.23 通过 Express 写了一个简单的 Web 应用。通过 node 指令执行该文件后，该应用将会通过 3000 号端口等待 Web 客户端的连接，并在客户端对其进行访问时，返回内容为 'Hello World' 的响应。

代码清单 22.23 Express 的简单代码示例

```
var express = require('express');
var app = express.createServer();

app.get('/', function(req, res) {
    res.send('Hello World');
});

app.listen(3000);
```

下面是对代码清单 22.23 中要点的说明。app.get 中的 get 与 HTTP 中的 GET 方法相对应。get 的第一个参数是 '/'，它对应于 URL 路径。也就是说，对于 http://localhost/ 这一主机名为 localhost 的 URL，它在接收了 GET 请求时，将调用该函数第 2 个参数所指定的回调函数。

回调函数将会接收两个参数。从代码清单 22.23 所使用的参数名称中也能看出，它们分别是表示请求的对象，及表示响应的对象。Express 程序设计的基本模式为，首先对请求 URL 及其内部回调函数的对应关系进行定义（这种对应关系被称为 URL 路由处理或 URL 分发处理），之后，通过回调函数获取请

求信息，并处理响应。

22.8.1 URL 路由

如代码清单 22.23 中的 app.get 所示，在 Express 的 URL 路由处理过程中，将会把 URL 路径传递给与 HTTP 方法名相对应的方法。此外，还有 app.post、app.put 及 app.del 等方法。其中需要注意的是，由于 delete 是 JavaScript 中的保留字，因此与 HTTP 中的 DELETE 方法相对应的方法是 app.del。而 app.all 则能够与所有的 HTTP 方法相匹配。这时，只要 URL 路径成功匹配，就能够确定所对应的回调函数。

我们可以像下面这样，通过形如 :foo 的方式来描述 URL 路径。在下面的例子中，/user/suzuki/ 这样的 URL 路径将会获得匹配。之后，可以通过 req.param.id 这样的形式来获取 URL 中 'suzuki' 这部分字符串。这种获取 URL 路径中部分元素的方式，在设计 RESTful 的 URL 时非常有用。

```
app.get('/user/:id', function(req, res) {
    res.send('user ' + req.paras.id);
// req.params.id 或 req.param('id') 可以表示 URL 字符串中的一部分 ( :id )
});
```

此外，还有一些其他的 URL 路径指定方式。可以同时列出多个 :foo 形式的元素匹配，也可以在匹配后添加 ?，以 :foo? 的形式，根据元素是否存在而进行匹配。如果使用 *，则会匹配所有情况。

```
/:foo/:bar
/:foo.:bar
/:foo/:bar?
/user/*
```

我们还可以通过正则表达式来实现更为复杂的 URL 路径匹配。对正则表达式的说明在此省略。

22.8.2 请求处理

包括前一节说明过的 URL 路径，Web 应用中请求处理的主要对象有 GET 请求中的查询参数，及 POST 请求中的表单数据。本节之后将介绍获取这些值的方法。

我们可以通过形如 req.query['foo'] 的方式来获取查询参数的值。假如请求 URL 为 http://localhost:3000/?foo=bar，则取得的值是 'bar'。假如请求 URL 如同 http://localhost:3000/?foo=bar&foo=bar2 这样，在查询参数中具有多个不同的值，req.query['foo'] 则会得到一个数组，其值为 ['bar', 'bar2']。

在通过 POST 方法发送 HTML 表单的输入值之后，该值将会由 HTTP 正文传递至 Web 应用。我们需要在 app.js 中书写以下代码，以实现对正文数据的分析，并将其分解为表单域名与域值（如果通过之后介绍的 scaffold 创建，则会自动生成这条代码）。

```
app.use(express.bodyParser());
```

我们可以通过形如 req.body['foo'] 的形式来获取表单内的表单域值。这里的 foo 就代表表单域名。和查询参数的情况相同，如果具有多个不同的值，则将获得一个数组。

如果像下面这样制定 HTML 表单的域名，则能够通过 req.body['user'] 获取一个形如 { name: 值 , email: 值 } 的对象。

```
<input type="text" name="user[name]" />
<input type="text" name="user[email]" />
```

而通过 req.param 方法，则可以以相同的方式，获取 URL 路径元素、查询参数及表单数据等所有内容。该方法可以以 req.parem('foo') 的形式使用。其中，第 1 个参数所接收的是 URL 路径的匹配名称、查询参数名，或表单域名。如果像 req.param('foo', 'default-value') 这样，指定了它的第 2 个参数，则会在方法没有返回值的时候返回该参数所指定的默认值。在本章的最后，将会介绍一个 req.param 方法的具体示例。

22.8.3　响应处理

表 22.14 列出了一些具有代表性的 Express 响应处理方法。其中最为重要的是 render 方法。之后的 22.8.5 节将对其进行说明。

表 22.14　具有代表性的响应处理方法

方法	说明
res.header(key, [val])	设置响应头部信息
res.sendfile(path[, options[, callback]])	通过响应正文返回指定文件路径的内容
res.send(body[, headers[, status]])	通过响应正文返回指定的字符串
res.redirect(url[, status])	向指定 URL 返回一个重定向响应
res.render(view[, options[, fn]])	将响应处理转让给指定的视图

22.8.4　scaffold 创建功能

在 Express 中还包含了 express 指令，该指令提供了 scaffold 创建功能。所谓 scaffold 创建功能，指的是通过 1 条指令来创建应用程序的框架文件。执行下面的指令之后，将会为一个名为 myapp 的应用程序创建其 scaffold。可以将 myapp 改写为任意的应用名称。

```
$ express myapp
$ cd myapp
$ npm install -d
```

在 myapp 目录下将会自动创建一个 app.js 文件。可以像下面这样执行该文件。默认情况下，这一 Web 应用将会在 3000 号端口等待连接。

```
$ node app.js
```

22.8.5　MVC 架构

通过 scaffold 创建功能生成的 app.js 文件包含下面这段代码。这段代码所执行的功能是 Express 中 MVC 架构的关键。

```
// app.js 中的一段代码
app.get('/', function(req, res) {
    res.render('index', {
        title: 'Express'
    });
});
```

res.render 方法的第 1 个参数是视图名，第 2 个参数是一个将被传递至视图的上下文语境对象。根据 MVC 中的习惯，这个被传递至视图的上下文语境对象被称为模型。不过本书将不会使用这一名称，而将其称为上下文语境对象。视图将会引用控制器所传递的上下文语境对象，并生成最终的输出结果。这便是 MVC 架构中视图的功能。在视图中，通常都会使用模板语言，下一节将会对其进行介绍。

■ MVC 中的各部分所承担的功能

在 MVC 中，创建上下文语境对象并将其传递给视图，是控制器的功能之一。如果将模型的功能定义为后端处理，控制器的功能则可以被定义为对模型与视图的对应关系的管理（关于模型，有着多种不同的定义方式，这里使用的仅是其中之一）。

在采用了 MVC 架构的 Web 应用中，控制器还具有其他一些功能。前文介绍过的 URL 路由功能正是其中之一。控制器的另一个功能是进行数据绑定处理，该处理可以将接收到的表单等数据转换为内部数据。MVC 相关的术语常常具有多种含义，如有兴趣，可以参考其他的书籍。

本节之后将继续介绍 res.render。在上面的代码中，第 1 个参数是一个值为 'index' 的视图名。该视图名与实际存在的文件相对应。我们可以在 app.js 中添加下面一段代码，以实现两者的关联处理。

```
app.set('views', _dirname + '/views');
app.set('view engine', 'jade');
```

在这样设定之后，'index' 就会在 Express 应用的根目录（app.js 的所在目录）中，与相对路径为 views/index.jade 的文件相关联。上面的 res.render 代码作用是将响应处理委托给 views/index.jade。顺便一提，在 Java Servlet 中，这种将响应处理委托给其他文件的做法被称为转发处理。

在此，我们简单总结一下 Web 应用中 MVC 架构的功能与相关执行方式。首先，控制器将会接收请求。根据 URL 路径等信息，控制器将会调用相应的内部处理模型。之后，将会把模型的调用结果作为上下文语境对象，传递给视图。视图从上下文语境对象中读取所需的值，并输出响应。

22.8.6 模板语言 Jade

在代码清单 22.23 中，我们在源代码中以硬编码的方式书写了响应字符串。如果希望输出 HTML，则必须对 HTML 字符串进行硬编码。然而，如果将这类用于响应的 HTML 字符串硬编码，则会降低代码的可维护性。如果能将和界面相关的代码（视图层代码）与其他部分相分离，则能够提高代码的可维护性。

在开发 Web 应用时，我们可以在视图层使用模板语言。在模板语言的代码结构中，将以最终输出的 HTML 中的不变部分为基础，并在其中嵌入执行时的可变部分。对于 Java 来说，JSP 是最为知名的模板语言。而对 PHP 来说，其本身就是一种模板语言。默认情况下，Express 所使用的模板语言是 Jade（也可以使用其他的模板语言）。可以从以下 URL 中获取有关 Jade 的相关信息。

http://jade-lang.com/

■ Jade 的规则

简单的 Jade 使用规则如下所示。

- **如果再行首写上标签，该语句将成为 HTML 元素**

```
p
->
<p></p>
```

- **接在标签之后的字符串则将成为 HTML 元素的内容。如果该字符串需要跨越多行，可以通过 "|" 将它们相连接**

```
p 内容
->
<p> 内容 </p>
```

```
p 内容
| abc
| xyz
->
<p> 内容 abcxyz</p>
```

- **如果写的是包含缩进的标签，该语句将会成为嵌套标签**

```
p
    span 内容
->
<p><span> 内容 </span></p>
```

- **而如果使用了冒号，则可以省略缩进**

```
p: span 内容
->
<p><span> 内容 </span></p>
```

- **我们可以以 CSS 选择器风格的书写方式，指定 id 属性与 class 属性（其中，div 属性可以被省略）**

```
p#foo 内容
->
<p id="foo">内容</p>
```

```
p.bar 内容
->
<p class="bar">内容</p>
```

```
p#foo.bar.baz 内容
->
<p id="foo" class="bar baz">内容</p>
```

```
#foo 内容
->
<div id="foo">内容</p>
```

- 我们可以通过在标签名后使用小括号，指定属性

```
a(href='/foo/bar') 内容
->
<a href='/foo/bar'>内容</a>
```

- 如果语句以 // 开始，则会以 HTML 的风格将其注释

```
//p#foo.bar.baz 内容
->
<!-- p#foo.bar.baz 内容 -->
```

- 我们可以通过 #{ 变量名 } 的形式，获取 JavaScript 中的变量的值（包括在 Jada 进行定义的变量，或者通过 res.render 的第 2 个参数所接收的上下文语境对象的属性）

  假设 JavaScript 中变量 title 的值为 'Express'，则有以下结果

```
p #{title}
->
<p>Express</p>
```

- 我们可以通过 " 标签名 = 变量名 " 或 " 属性名 = 变量名 " 的形式，获取 JavaScript 中的变量的值（还可以使用字符串连接表达式）

```
p= title
->
<p>Express</p>
```

```
a(href = title)= title
->
<a href="Express">Express</a>
```

```
a(href = '/' + title)= title
->
<a href="/Express">Express</a>
```

- 如果在行首写有 "–"，则能够在之后使用 JavaScript 代码

```
- var foo = 'bar';    // 为变量赋值

- for (var key in obj)
p = obj[key]

- if (foo)
ul
    li = foo
    li 内容
- else
p 内容

- var items = ["one", "two", "three"]
- each item in items
  li = item
```

22.8.7　MongoDB（数据库）

通常来说，如果 Web 应用达到了某一规模，则需要在后端使用数据库。当前，使用 MySQL 等 RDBMS 是最为普遍的做法。

Node.js 有多个可以对 RDBMS 进行操作的包。在执笔本书时，尚无某种事实标准，还不能断定哪一个包将会成为之后的主流。因此，这里我仅向读者介绍下面的 nodejs-db。

http://nodejsdb.org/

本书将不会介绍 RDBMS，而是会对 Node.js 中 MongoDB 的操作方法进行说明。MongoDB 是一种所谓的 NoSQL。下面是 MongoDB 官方站点的 URL。

http://www.mongodb.org/

在 Node.js 中，有多个包可以对 MongoDB 进行操作。关于这些包的信息，请参见下面的 URL。

http://docs.mongodb.org/ecosystem/drivers/node-js/[①]

本书使用了下面站点所提供的 Mongoose 包，以实现在 Node.js 中对 MongoDB 进行操作。

http://mongoosejs.com/

可以像下面这样安装 Mongoose。

```
$ npm install mongoose
```

MongoDB 的安装方法在此省略。可以向下面这样通过 mongod 指令来启动 MongoDB。

```
$ mkdir data
$ mongod --dbpath data
```

mongod 是 MongoDB 的服务器进程。在 MongoDB 中包含了一个客户端工具，它具有一个名为 mongo 的会话式 JavaScript 壳层。该壳层还有另外一个作用。在启动了 mongo 指令后，能够方便地确认 MongoDB 的运行情况。该方法的实质是在 Node.js 应用中操作 MongoDB 数据时，通过 mongo 指令，以会话的形式确认结果。如果在同一台 PC 上执行了 mongo 指令和 mongod 指令，则会在启动该工具后自动连接 mongod 服务器。

■ MongoDB 的概念

本书将不会详细介绍 MongoDB。不过，如果大家不了解基本术语，就无法理解进一步的说明。本节对相关术语进行了总结（表 22.15）。

可以将 MongoDB 直观地理解为一种 JavaScript 对象的持久化形式，这种说法虽然不严密，但能够帮助理解 MongoDB 的使用方式。持久化之后的对象与表 22.15 中的文档相对应。对象的属性对应于其中的域。可以通过 ID 识别文档，ID 会被作为键来进行文档的获取（检索）、更新与删除操作。这些是 MongoDB 的基本功能。

不过如果仅支持这些功能，MongoDB 也就与 KVS（键值存储）没有区别了。在 MongoDB 中，可以通过域值接受查询，实现更复杂的面向文档数据库操作。与 RDB 形式的 SQL 相比，MongoDB 中查询语句的表达能力较弱，但相应地，得益于分布式处理，MongoDB 能够发挥出较高的可扩放性能。

表 22.15　MongoDB 的组成元素

名称	说明
数据库	聚集的容器
聚集	文档的容器。相当于 RDB 中的表。在数据库中通过唯一的名称标识
文档	属性的集合。在聚集内通过唯一的 ID 标识。其内部结构为 BSON（Binary JSON）格式
域	名称与值的配对。值的类型基于 JSON 标准。在文档内通过唯一的名称标识

① 原书的链接是较早的版本，此处已更新为新版。——译者注

22.8.8　Mongoose 的实例

MongoDB 是一种不含架构的面向文档数据库。在使用 MongoDB 时，我们无需像使用 RDB 那样实现对数据库表进行设计（对架构进行定义）。不过习惯上，在 Mongoose 中仍会书写架构定义。对于这种设计方式的褒贬不一，本书不作评价。

代码清单 22.24 通过 Mongoose 在 MongoDB 中新建了一个文档。前一节也提到过，在直接将 JavaScript 对象传递给 save 方法之后，MongoDB 将会保存该对象。而对象的属性值则会直接被转换为 MongoDB 文档的域值。

代码清单 22.24　在 MongoDB 中新建文档

```
var mongoose = require('mongoose');

// mydb 是数据库名称（可以选用任意喜欢的名称）
mongoose.connect('mongodb://localhost/mydb');

var Schema = mongoose.Schema;

// 定义数据库架构
// 'articles' 是聚集名（可以选用任意喜欢的名称）
var Articles = mongoose.model('articles', new Schema({
    title : String,
    body : String,
    date : { type : Date, default : Date.now }
    })
);

// 新建文档
var obj = new Article();
obj.title = 'hello';
obj.body = 'hello body';

// 保存
obj.save(function(err) {
    if (err) throw err;
});
```

在执行了代码清单 22.24 之后，将会显示如图 22.2 的界面，可以通过 mongo 的会话式壳层确认。在 MongoDB 中，我们不需要事先准备数据库和聚集。如果它们原本并不存在，则会被自动创建。

图 22.2　通过 mongo 指令确认操作

```
$ mongo
MongoDB shell version: 1.8.2
connecting to: test

// 显示数据库一览（在执行代码清单 22.24 之前，仅存在 admin 与 local 两个）
> show dbs
admin (empty)
local (empty)

// 在执行代码清单 22.24 之后，将会自动创建数据库 mydb
> show dbs
admin (empty)
local (empty)
mydb 0.0625GB

// 切换当前数据库
> use mydb
switch to db mydb

// 显示聚集一览（在执行了代码清单 22.24 之后，将会自动创建聚集 articles）
```

```
> show collections
articles
system.indexes

// 显示文档一览（在执行了代码清单 22.24 之后，将自动创建一个文档）
> db.articles.find()
{ "data" : ISODate("2011-07-03T13:02:26.483Z"), "_id" : ObjectId("4e106862aea
4e61126000001"), "title" : "hello", "body" : "hello body" }
```

下面的代码将通过 Mongoose 来显示 MongoDB 中的文档一览（代码清单 22.25）。find 方法的第 1 个参数将接收搜索条件。如果传递的是 { title:'hello' }，则将以与其完全一致为条件来搜索域值，如果传递的是 { title: /^h.*/ } 这样的正则表达式，则将以正则表达式匹配为条件搜索。像代码清单 22.25 这样传递空对象的话，则会返回聚集内的所有文档。关于其他可用的搜索条件的详细信息，请参见 MongoDB 的参考文档。

find 方法（依然）是一种异步 API。它的第 2 个参数接收的是一个回调函数。而回调函数的第 2 个参数将会传递与搜索条件相匹配的文档。MongoDB 文档基本上都是通过这种形式与 JavaScript 对象相关联的，所传递的对象将会以数组的形式表示。

代码清单 22.25　显示 MongoDB 的文档一览

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/mydb');

var Schema = mongoose.Schema;
var Article = mongoose.model('articles', new Schema({
    title : String,
    body : String,
    date : { type: Date, default: Date.now }
    })
);

Article.find({}, function(err, docs) {
    docs.forEach(function(doc) {
        if (err) throw err;
        console.dir(doc);
    });
});
```

22.8.9　使用了 Express 与 Mongoose 的 Web 应用程序

■ 文档管理应用程序

在本章的最后，我们将会尝试编写一个使用 Express 与 Mongoose 的 Web 应用。不过，由于篇幅所限，其结构将非常简单。该 Web 应用仅含有文档一览及文档创建功能，所创建的文档将会保存于 MongoDB 中。

首先需要像下面这样，通过 Express 的 scaffold 创建功能来获取应用的基本框架。可以使用任意的应用名称（这里使用了 myapp）。

```
$ express myapp
$ cd myapp
$ npm install -d
$ npm install mongoose
$ mkdir models
```

■ 创建模型

下面将会在 app.js 的相对路径 models 下创建一个名为 article.js 的文件（代码清单 22.26）。它的功能相当于 MVC 中的模型。模型所提供的功能与 Mongoose 所提供的功能相同。如果需要添加自定义功能，则需要自行在 Article 类中添加方法（根据 JavaScript 的习惯，通常会添加至 Article.prototype 中）。

代码清单 22.26　models/article.js

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/mydb');

var Schema = mongoose.Schema;

var Article = mongoose.model('articles', new Schema({
    title : String,
    body : String,
    date : { type: Date, default: Date.now }
    })
);

module.exports = Article;
```

为了能在 app.js 中使用代码清单 22.26 中的 article.js 模块，需要在 app.js 中添加以下代码。

```
var Article = require('./models/article');
```

这样一来，就能够在 app.js 中使用 Article 对象（Article 类）了。

■ 提供文档一览功能的控制器

代码清单 22.27 修改了 app.js 接收到发送自 / 的请求时所执行的代码。该 URL 将会执行显示文档一览的操作。其基本运行结构依然是调用 res.render。所使用的 Jade 文件也仍然保持 index.jade 即可。不过传递给视图的上下文语境对象则发生了变化。在 Jade 文件中，我们可以通过 docs 的名称引用 MongoDB 的文档数组。

代码清单 22.27　提供文档一览功能的控制器（app.js）

```
app.get('/', function(req, res) {
    Article.find({}, function(err, docs) {
        if (err) throw err;
        res.render('index', { title: 'Document list', docs: docs });
    });
});
```

■ 提供文档一览功能的视图

需要像代码清单 22.28 那样修改 views/index.jade 文件，以通过 HTML 表格来显示文档一览。在表格中，包含了文档标题及正文的表格列，在标题列中将列出文档的链接，以供之后的文档显示。文档显示链接的路径格式为 "/ 文档 ID"。

代码清单 22.28　views/index.jade

```
table(border='1')
    - for (var i = 0; i < docs.length; i++)
     tr
        td: a(href = '/' + docs[i]._id) = docs[i].title
        td = docs[i].body
```

■ 文档显示功能

在 app.js 中写有控制器接收到 '/ 文档 ID' 路径格式的请求时的执行代码[①]。为此我们需要将代码清单 22.29 中的内容添加至 app.js 中。可以通过 req.param('id') 来获取请求 URL 中路径元素的值，并将其作为 Mongoose 中 findOne 方法的搜索条件。show.jade 文件在此省略。

① 该 URL 路径也会对 /favicon.ico 请求进行相应，对此请加以注意。

代码清单 22.29 用于文档显示的控制器（app.js）

```
app.get('/:id', function(req, res) {
    Article.findOne({ _id: req.param('id') }, function(err, doc) {
        if (err) throw err;
        res.render('show', { title:'Document', doc: doc });
    });
});
```

■ 文档创建功能

接下来，我们将编写文档的创建界面。这里的请求 URL 使用了 '/create' 这样一个稍有些特别的名称（代码清单 22.30）。由于代码清单 22.29 中的 '/:id' 涵盖了这种情况，因此需要在它之前书写下面的代码[①]。

代码清单 22.30 用于文档创建的控制器（app.js）

```
app.get('/create', function(req, res) {
    res.render('create', { title:'Create' });
});
```

同时，我们需要像代码清单 22.31 这样编辑 create.jade 文件。在文件中准备了用于书写标题及正文的域，并将 POST 目标的 URL 设定为了 '/'。由于可以通过 '/' 来显示文档一览，因此可以将该路径视为文档的容器。对该容器 POST 文档之后，就会创建一个新的文档。这也是 RESTful 风格 URL 设计模式的习惯做法。

代码清单 22.31 文档创建的表单界面（views/create.jade）

```
form(action = '/', method = 'POST')
    p Title
    input#title(name = 'title', type = 'text')

    p Body
    textarea#body(name = 'body', cols = '40', rows = '10')

    input#submit(type = 'submit', value = 'Save')
```

而 POST 目标的控制器则可以按代码清单 22.32 的方式编写。对文档保存操作的响应使用重定向处理，是 Web 应用开发中的标准做法。

代码清单 22.32 用于文档保存的控制器（app.js）

```
app.post('/', function(req, res) {
    var doc = new Article();
    doc.title = req.param('title');
    doc.body = req.param('body');
    doc.save(function(err) {
        if (err) throw err;
        res.redirect('/');
    });
});
```

本节介绍的代码没有进行任何错误处理，请加以注意。这里我们试图以尽可能短的代码说明执行方式。

为了实现所谓的 CRUD（Create、Read、Update 及 Delete）处理，还必须为应用添加文档的更新与删除功能。在本书中将省略这部分内容。Mongoose 的 save 方法可以创建一个新的文档，并以指定的文档 _id 属性值保存。如果文档已经存在，会对其进行更新。删除操作则可以通过 remove 方法来实现。了解了这些方法之后，添加新功能也并非是一件难事。

① 这种 URL 设计是存在问题的。事实上，URL 设计是一项困难的工作。在如今的 Web 应用中，URL 设计是最为重要的接口设计，因此在实际的应用开发中，必须慎之又慎。

后　记

在学习 IT 新技术时，我往往会患得患失，考虑投入与回报的比例。我虽不喜欢这样的自己，但到了一定年龄之后，我确实无法否认，自己希望学习的是投资回报率更高的技术。当然，不考虑技术的流行程度，仅凭兴趣学习，也并非坏事。我反而觉得这类人更加值得尊敬。但我并不强求所有人在任何时候都必须持有这种态度，以高涨的热情来学习新技术。

倘若纯粹考虑学习的投资效率，抽象度较高的知识通常会更有长远价值。如果该技术与特定的语言或平台无关，投资效率也就会更高。然而我认为，技术人员不应该仅专注于新知识的学习。虽然不必无故做出与时代脱节的姿态，但还是应该与时代及流行风潮保持一定的距离。能够脱颖而出成为主流的技术，即使其中有偶然成分，也一定会有其必然性。不能总是对主流技术持以疏远的态度，在适当的时候，也应该遵从主流，学习主流技术。

话虽如此，我长久以来都不太关心主流技术，因此这么说有些底气不足。

学习 JavaScript 的投资效率其实相当不错。这是因为用不了多久，JavaScript 可能就会成为互联网中引领各种技术发展的标准语言。此外，JavaScript 虽然是一种主流的语言，却有不少令人意外的古怪之处。因此，即使对流行的 Java 语言不感兴趣，也很可能会被 JavaScript 所吸引。尽管 JavaScript 相关的技术大多很简陋，但这倒也是一种独特的魅力。

最后，要向给本书提供了大力支持的内田大嗣表示感谢。虽然无法在本书中完全体现出他近乎狂热的挑刺功力，但得益于他的独到眼光，本书的内容才能有如此深度。

井上诚一郎

如今，开发者能够使用 JavaScript 写出各种类型的应用程序。最初它主要用于增强浏览器的使用体验，而现在，服务器端应用（Node.js）、Linux 下的 GUI 应用（Gjs、Gnome Seed），甚至 iPhone 及 Android 的应用程序（Titanium Mobile）都可以通过 JavaScript 来实现。说得夸张些，只要学会了 JavaScript 程序设计，就能够在计算机中实现任何希望实现的功能。JavaScript 给人的印象已经发生了巨大的变化，它不再是那种仅被用于禁止右键点击，或是在状态栏中显示文本的简单语言。JavaScript 现在正处于流行之巅。尽管这门语言基于原型的特性以及 this 引用的用法十分复杂，仍然无法阻挡 JavaScript 的流行。

JavaScript 是如此流行，让人不禁觉得，在将来的义务教务阶段，也许会要求学生们具备 JavaScript 的读写能力。继日语、英语之后，JavaScript 会成为第三种人皆可读的语言[①]。真是这样的话，未来将会变得令人期待。孩子们将能够使用跨站点脚本语言，在早晨通过类似于 alert('Good morning'); 的方式互相打

① 虽然程序设计语言会被纳入义务教育的说法并非第一次出现，但目前来看，这样的未来还很遥远。——译者注

招呼。那时，要是大人无法回答孩子们关于 JavaScript 的疑问，可就颜面无存了。为了能让自己成为受孩子尊敬的人，必须学会 JavaScript。总之，学好 JavaScript，总是会有些好处的。

土江拓郎

JavaScript 可以说是我最先学会的一门语言。在当初加入 Ariel 公司时，自己几乎什么都不会。离开公司后不久，本书的共著者井上诚一郎对我要求道："滨边，你去学习一下 Ajax 吧。"那时候正是 Ajax 最为流行的时期，人们还并不知道 JavaScript 的人气能够维持至今，但仍然不减对 JavaScript 的热情。凭借这样的势头，JavaScript 至今仍然非常流行。现在回想起来，要不是有当时他的那句话，我现在不可能成为一个 JavaScripter 吧。借此机会，要向井上表达我的谢意。

如今，仅凭 JavaScript 就能实现大量功能。在 Web 应用程序、原生应用程序、智能手机及智能电视应用，以及客户端 / 服务器端程序中，都可以使用 JavaScript。在这种情况下，要是只对在 Web 中搜索的代码片段或是库稍作修改，插入自己的代码，显然是无法适应如此广泛的应用的。对于 JavaScript 的初学者来说，必须要以某种方式来系统地学习 JavaScript 的知识。如果本书能够帮助你成为更高一级的 JavaScripter，我将不胜荣幸。

滨边将太

索引

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

微博 @图灵教育 : 好书、活动每日播报
微博 @图灵社区 : 电子书和好文章的消息
微博 @图灵新知 : 图灵教育的科普小组
微信 图灵访谈 : ituring_interview，讲述码农精彩人生
微信 图灵教育 : turingbooks